The VLDB Journal (1998) 7: 48–66

The VLDB Journal

Data partitioning and load balancing in parallel disk systems

Peter Scheuermann[1], **Gerhard Weikum**[2], **Peter Zabback**[3]

[1] Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60208, USA; E-mail: peters@eecs.nwu.edu
[2] Department of Computer Science, University of the Saarland, P.O. Box 151150, D-66041 Saarbrücken, Germany; E-mail: weikum@cs.uni-sb.de
[3] Tandem Computers Incorporated, 10100 North Tantau Avenue, Cupertino, CA 95014-2542, USA; E-mail: zabback@loc251.tandem.com

Edited by W. Burkhard. Received May 17, 1994 / Accepted June 9, 1997

Abstract. Parallel disk systems provide opportunities for exploiting I/O parallelism in two possible ways, namely via inter-request and intra-request parallelism. In this paper, we discuss the main issues in performance tuning of such systems, namely striping and load balancing, and show their relationship to response time and throughput. We outline the main components of an intelligent, self-reliant file system that aims to optimize striping by taking into account the requirements of the applications, and performs load balancing by judicious file allocation and dynamic redistributions of the data when access patterns change. Our system uses simple but effective heuristics that incur only little overhead. We present performance experiments based on synthetic workloads and real-life traces.

Key words: Parallel disk systems – Performance tuning – File striping – Data allocation – Load balancing – Disk cooling

1 Introduction: tuning issues in parallel disk systems

Parallel disk systems are of great importance to massively parallel computers since they are scalable and they can ensure that I/O is not the limiting factor in achieving high speedup [10, 55, 63]. However, to make effective use of the commercially available architectures, it is necessary to develop intelligent software tools that allow automatic tuning of the parallel disk system to varying workloads. The choice of a striping unit is an important parameter that affects the response time and throughput of the system. Equally important is the decision of how to allocate the data on the actual disks and how to perform redistribution of the data when access patterns change, especially when the load becomes imbalanced across the disks due to skewed access frequencies. These tuning options need to be performed dynamically, using simple but effective heuristics that incur only little overhead.

This paper presents a set of performance-tuning techniques for parallel disk systems. These techniques are orthogonal to the techniques for high availability that typically are employed in parallel disk systems (e.g., RAID levels) and can be applied to a wide spectrum of applications ranging from conventional file systems and WWW servers to database systems. Throughout the paper we assume that the underlying computer architecture is that of a shared-memory multiprocessor; extensions to distributed-memory architectures are conceivable but are not considered in this paper.

In order to effectively exploit the potential for I/O parallelism in parallel disk systems, data must be partitioned and distributed across disks. The partitioning can be performed at two levels:

1. The physical (block or byte) level. The term *striping* is used for this variant of partitioning schemes which divides a file into fixed-size runs of logically consecutive data units that are assigned to disks in a round-robin manner [43, 51, 65, 68]. The *striping unit* denotes the number of consecutive data bytes or blocks stored on a given disk.
2. The application level. The term *declustering* has been employed in relational database systems to denote partitioning schemes that perform a horizontal division of a relation into fragments based on the values of one or several attributes. Among the schemes employed for single attribute partitioning are hashing and range partitioning [18, 27], while techniques based on Cartesian product files have been advocated for multiple attribute declustering (e.g., [20, 21, 29, 44]).

Striping has an advantage over application-level methods in that it can be applied as a generic low-level method for a wide spectrum of data types (all of which are ultimately mapped into block-structured files). In this paper, we therefore restrict our attention to data partitioning via file striping. In the following, a file may denote a table space or index space in a relational database, a logical object cluster in an object-oriented database, a document such as WWW pages in a multimedia information system, or, indeed, simply a Unix-like sequence-of-blocks file. We shall use the term *striping width* of a file to denote the number of disks over which a file is spread. A logically consecutive portion of the file that resides on one disk and whose size is a striping unit is called a *run*. All runs of a file that are mapped to the same disk are combined into a single allocation unit called an *extent*.

Striping provides opportunities for exploiting I/O parallelism in two possible ways. Intra-request (intra-operation) parallelism allows the parallel execution of a single request by multiple disks. Inter-request (inter-operation) parallelism can be achieved if independent requests are being served by the disk system at the same time. The *degree of parallelism* in serving a single data request is the number of different disks on which the requested data resides.

1.1 Tuning issues in data partitioning

The striping unit is an important parameter that must be chosen judiciously in order to reduce the service time of a single request or to improve the throughput of multiple requests [12, 13, 25, 33, 48, 80]. A large striping unit tends to cluster a file on one disk, which does not allow any degree of intra-request parallelism. In consequence, the service time of a request is not improved, but the throughput is increased if the requests are uniformly distributed across all disks. At the other end of the spectrum, a small striping unit provides very good response time for a light load, but severely limits the throughput, as the total amount of device-busy time consumed in serving a single request increases with a decreasing striping unit. Consequently, for small striping units, the response time may deteriorate under a heavy load due to queueing delays. In practice, it is necessary to choose the striping unit such that a certain objective function is minimized. One such objective function aims at minimizing the response time subject to a constraint on the achievable throughput.

In [11, 12, 48] heuristic methods are proposed to determine the striping unit of a disk array based on the knowledge of the average request size and the application's expected multiprogramming level under the assumption of a closed queueing model. While these assumptions may be valid for relatively small multiprogramming levels, they do not scale up to data management systems with large numbers of concurrent users, which translates to high arrival rates with stochastic load fluctuation. For such systems, it is most crucial to guarantee a certain level of performance during peak periods.

Most heuristic methods also advocate choosing a global striping unit, i.e., the same striping unit for all files in the system [11,12,48]. However, many applications such as multimedia information systems (e.g., in digital libraries or medical applications) exhibit highly diverse file characteristics, making it desirable to be able to tune the striping unit individually for each file. Furthermore, file-specific striping allows incremental repartitioning of files. Consider, for example, the case where a global striping unit may be appropriate at some moment in time, but later on the overall load is increasing so that the response time becomes critical. File-specific striping enables us incrementally to restripe only the most frequently used files to a larger length of striping unit, thus reducing the disk utilization and thereby decreasing the response time under a high load, while leaving the other files with the (old) global striping unit.

After the striping unit has been determined (globally or on a file-specific basis), the file system must derive the striping width, i.e., the number of disks across which the file(s) is (are) spread. In our model, a file is either spread across all disks, or, if the file is relatively small, its width is obtained as the quotient of the file size and the striping unit. Similar response time constraints to the ones discussed above may justify that some files be stored on a "dedicated" subset of disks in order to avoid contention,and hence their striping width would be limited by the number of disks in this subset. This type of constraint is not pursued in this paper.

1.2 Tuning issues in data allocation and load balancing

The striping unit(s) and striping width(s) are only some of the parameters that affect the response time or throughput of a parallel disk system. The decision of how to allocate the files on the actual disks is equally important in order to obtain good *load balancing*. Load refers to the amount of work done by each disk and it affects both the response time and throughput. Balancing the load contributes towards minimizing the average the queues associated with the disks (minimizing the service time variance per disk could be another optimization criterion [49], which is not considered in this paper). Very small striping units lead to very good load balancing; in the extreme case each request involves all the disks in the system so that the load is perfectly balanced. But throughput considerations require for many applications that we choose large striping units (e.g., the size of a cylinder) [11–13, 33, 48, 51, 80]. Thus, load balancing needs to be performed, even if striping is employed.

Load balancing is particularly challenging for evolving workloads, where the hot (i.e., frequently accessed) and cold (i.e., infrequently accessed) files (or portions of them) change over time. Such situations only can be counteracted by reallocating some of the data, and such reorganizations should be performed online without requiring the system to be quiescent. In order to perform this desired form of adaptive disk load balancing, it is necessary to dynamically estimate the frequencies of the requests to the various files or data partitions, as well as the request sizes. To account for these parameters, the file system must keep track of the following related statistics:

- the *heat* of extents and disks, where the heat is defined as the sum of the number of block accesses of an extent or disk per time unit, as determined by statistical observation over a certain period of time,
- and the *temperature* of extents, which is defined as the ratio between heat and size [14, 16, 42].

If the striping unit is a byte and all files are partitioned across all disks in the system, then we obtain a well-balanced I/O load. While this approach may be adequate for supercomputer applications characterized by very large request sizes [41] (i.e., a high data rate), it certainly limits the throughput of transaction processing applications characterized by a high rate of small random read and write requests [33] (i.e., a high I/O rate). As soon as the striping unit is relatively large (e.g., a track or cylinder), the need for load balancing reappears immediately, even if the files are partitioned across all disks. This is due to the fact that the heat of the various blocks or extents is often distributed in a highly non-uniform manner.

1.3 Contribution and outline of the paper

This paper addresses several key issues in the automatic tuning of parallel storage systems. Various aspects of earlier versions of our approach have been published in [71,72,79,80], and [78,84] give an account of the COMFORT project where this work has been applied. In this paper, we describe automatic tuning methods that are more advanced than the those used earlier and we emphasize their interaction in an actual system. We provide guidelines for potential system architects of self-reliant storage systems, and we give comprehensive experimental evidence of the viability and benefits of our approach.

The specific contributions of this paper are the following:

1. We derive and evaluate an approximative, analytic model for choosing a near-optimal global striping unit for all files in the system. Our model aims at minimizing the mean response time of all file access requests under the assumption of a perfect load balance. The optimization is performed for a given workload that is characterized by the average request arrival rate and the mean request size over all files. This analytic model determines a degree of parallelism that attempts to minimize the multi-user response time; given this degree of parallelism, the striping unit and striping width easily can be tuned. The model and the derived data partitioning method are significant extensions to the method outlined in [79]. In particular, our new algorithm explicitly takes into account queueing delays by providing a computationally tractable analytical approximation for the underlying fork-join queueing model [62]. In contrast to the modeling approach of [48] that uses a closed queueing model, our method is based on an open queueing model that seems more appropriate for systems with many clients (e.g., Web servers or database applications).
2. Our model for determining the performance impact of a global striping unit is complemented by additional considerations that allow us to choose the striping unit on a file-specific basis. In this case, we also consider the impact of the average request size of a particular file on the mean service time of requests to this file. While this extension has the potential for additional performance benefits, it turns out experimentally that these extra gains are minor. However, we consider the capability of supporting file-specific striping units an important benefit in dealing with files that need to be reorganized individually when the stationary load increases or when the number of disks changes. For example, hot files may have a striping unit different from that of cold files (see also [81] for similar considerations on flexibility and dynamic adaptation in parallel disk systems).
3. We present an online heuristic method for dynamic load balancing referred to as "disk cooling". This method counteracts load imbalances that are due to skewed access frequencies by performing online data migration steps from hot onto cold disks; hence the name "disk cooling". As our experiments demonstrate, this method achieves significant performance gains compared to methods that rely solely on static data placement for load balancing (e.g., round-robin placement of track-sized striping units). Our disk cooling method can cope with evolving workloads where the hot and cold portions of the data change over time. This is achieved by employing a heat-tracking method based on moving-average values which is very responsive to sudden changes in heat. Furthermore, our dynamic load-balancing procedure is invoked automatically and the reorganization requests are treated as lower priority requests that occur concurrently with regular file access requests.
4. We show through various performance experiments the synergetic effects of combining our heuristic methods for data partitioning and disk cooling. Although the problems of data partitioning and load balancing are orthogonal issues, they are not independent, since the near-optimal choice of the striping units is done under the assumption of a perfectly balanced system. The performance experiments reported in this paper clearly illustrate the combined effects of these two methods, as well their advantages over conventional striping methods based on physical device units (e.g, block, track).

The remainder of this paper is organized as follows. We describe in Sects. 2 and 3 the main components of an intelligent file manager for parallel disk systems that performs automatic data partitioning, data allocation, and load balancing by incremental reorganization steps. In Sect. 4, we report on performance studies of our file system based on synthetic workloads and real-life traces. We conclude in Sect. 5 with a discussion on future extensions to our work.

2 Data partitioning

We have developed an approximative analytic model that aims at determining the optimal striping unit and the striping width on an individual file basis or on a global basis. These parameters can be chosen for each file individually, based upon the file's estimated average request size R, or globally by using instead the average request size over all files, denoted by $\overline{R}$. In either case, the optimization can be carried out in one or two phases, A and B, depending upon the anticipated arrival rate of requests. For low arrival rates of requests, where we can assume that no queueing delays occur, Phase A chooses a degree of parallelism that aims at minimizing the service time of an average request of size R (or $\overline{R}$) which is equivalent to minimizing the response time if the system operates in single-user mode. Phase B chooses a degree of parallelism that aims at minimizing the (multiuser) response time subject to the constraint that the achievable throughput is at least as high as the application's average arrival rate of requests to all files, denoted by $\overline{\lambda}$. The degree of parallelism, P_{eff}, chosen for an average request is then adjusted by choosing the minimum (normalized as explained below) between the outcomes of Phases A and B. The striping unit and width are then derived from the chosen near-optimal degree of parallelism, P_{eff}.

Our Phase B optimization uses an open queueing model in order to take into account explicitly the throughput considerations and queueing delays, and uses approximations to find a near-optimal degree of parallelism. As mentioned before, the striping method proposed by Chen et al. [12,48] is based on a closed queueing model. There, a heuristic formula is derived from experiments as well as approximative analytical treatment, which suggests a global striping unit of

$$SU = \sqrt{\frac{(L\ X\ (M-1)\ \overline{R}}{D}}\,, \tag{1}$$

where L is the average latency (sum of seek and rotational delays) of a disk, X is the transfer rate of a disk, M is the multiprogramming level of the application, $\overline{R}$ is the average request size, and D is the number of disks. Chen and Lee [11] further extend this approach by considering the impact of parity writes in a RAID level 5 system. Chen et al. [12, 48] also discuss the difficulty of estimating the multiprogramming level. As we pointed out earlier, we consider an open queueing model to be more appropriate for a data management system with a large number of users (as opposed, for example, to a file server in a LAN of workstations).

2.1 Phase A: minimizing service time

Given a number of files to be allocated, Phase A determines the optimal partitioning on a file-specific (global) basis based on the average request request size R ($\overline{R}$). This estimate can be derived in many cases from the file's type information. For example, in an OLTP system such as airline reservation or phone call switching and accounting, one can typically expect an average request size of a block. On the other hand, in multimedia applications such as digital libraries or medical archiving, we can expect that all requests will require access to an entire document (e.g., an image), and hence R would be the file size. A general discussion of how to determine such workload parameters for arbitrary applications is beyond the scope of this paper. Note, however, that it is trivial to determine the parameters in retrospect after having observed the access patterns for some time; then the presented tuning method can always be used for reorganizing existing files.

Let P be the degree of parallelism involved in serving an average request of size R, i.e., the number of disks involved in serving this request. In the absence of queueing delays, the expected service time, to be denoted by $T_{serv}(R, P)$, is in fact equal to the expected response time, to be denoted by $T_{resp}(R, P)$. The expected service time is given by

$$T_{serv}(R, P) = \max_i(t_{seek,i} + t_{rot,i}) + t_{trans}(R, P)\,, \tag{2}$$

where $t_{seek,i}$ and $t_{rot,i}$ $(i = 1, ..., P)$ denote the seek time and rotation time, respectively, of disk i involved in serving the request. For tractability purposes, we replace the right-hand side by the following approximation:

$$T_{serv}(R, P) = \max_i(t_{seek,i}) + \max_i(t_{rot,i}) + t_{trans}(R, P)\,. \tag{3}$$

Thus, we note that the solution to Eq. 3 in fact provides an upper bound for $T_{serv}(R, P)$.

In order to obtain approximate distributions for $T_{seek} = \max_i(t_{seek,i})$ and $T_{rot} = \max_i(t_{rot,i})$ we make the standard assumption that the delays at each disk, i.e., seek times and rotation times are independent and identically distributed random variables [8, 45]. In addition, we assume that the delay probabilities are unconditional, i.e., the probability of a delay does not depend upon the probabilities of previous delays. In reality, there may be a certain degree of correlation among these variables, for example in the case of a synchronized disk array where all disks heads move in tandem. Also, in some applications, it is possible to have a sequence of requests to successive blocks on a disk; in other words the probability of a seek distance is conditional upon the probabilities of previous seek distances.

Let us denote by $d_{seek,i}$ and $d_{rot,i}$ $(i = 1, ...P)$ the dual random variables that give us the distances traveled on disk i by the head or arm, respectively, from the current location to the requested one. We shall compute first the expected values of $D_{seek} = \max_i(d_{seek,i})$ and $D_{rot} = \max_i(d_{rot,i})$ and use these values to derive the expected values of T_{seek} and T_{rot}.

Under the assumptions given above, the cumulative distribution functions for D_{seek} and D_{rot} can be computed, respectively, as the product of P cumulative distribution functions of the random variables, $d_{seek,i}$ and $d_{rot,i}$, corresponding to the P disks involved in serving the request.

We compute first the probability mass function of $d_{seek,i}$:

$$Prob[d_{seek,i} = z] = \frac{2(C-z)}{C^2}\,, \tag{4}$$

where C denotes the number of cylinders on one disk. From this, we obtain the cumulative distribution function:

$$Prob[d_{seek,i} \leq z] = 1 - (1 - \frac{z}{C})^2\,. \tag{5}$$

It was shown in [8] that the expected value $E[D_{seek}]$ is given by :

$$E[D_{seek}] = C(1 - \prod_{i=1}^{P} \frac{2i}{2i+1})\,. \tag{6}$$

The product in Eq. 6 can be approximated by the following expression, with constants $a = 0.577$ and $b = -0.118$:

$$E[D_{seek}] = C(1 - a - b\ln(P))\,. \tag{7}$$

From here, the expected value of T_{seek} can be approximated by the following linear equation with appropriate (disk-type-dependent) constants e and f:

$$E[T_{seek}] = eE[D_{seek}] + f\,. \tag{8}$$

We note here that the equation which converts seek distance in cylinders to seek time in fact consists of two components, a non-linear one and a linear one [8, 67]. However, in our model, we are interested only in the expected values of the seek distance and the seek time, and the expected value of the seek distance lies in the linear part of the distance-time equation.

The rotation distance on a given disk i, $d_{rot,i}$, gives the fraction of a full rotation that is necessary to position the arm on the first block of the current request. In order to compute $E[D_{rot}]$, we make the common assumption that $d_{rot,i}$ is a uniformly distributed variable in the range $[0, 1]$, thus

$$Prob[d_{rot,i} \leq r] = r\,. \tag{9}$$

From this, we obtain the cumulative distribution function of $D_{rot} = \max_i(d_{rot,i})$ as

$$Prob[D_{rot} \leq r] = \prod_{i=1}^{P} Prob[d_{rot,i} \leq r] = r^P \,, \tag{10}$$

and furthermore $E[D_{rot}] = \frac{P}{P+1}$. It follows that the expected value of T_{rot} is given by

$$E[T_{rot}] = \frac{P}{P+1} ROT \,, \tag{11}$$

where ROT denotes the rotation time of a disk.

It can be seen from these equations that as the degree of parallelism, P, increases, both the expected seek time, $E[T_{seek}]$, and the expected rotation time, $E[T_{rot}]$, increase also. For small requests, these two components of service time are the dominant ones, hence the service time increases also. The only component of the service time that decreases with an increased degree of parallelism is the transfer time $t_{trans}(R, P)$. Each disk transfers $\frac{R}{P}$ blocks (assuming, for simplicity, identical subrequest sizes on the disks)), and if we ignore cylinder and head switches, the transfer time can be estimated as $\frac{R}{P}\frac{ROT}{B}$, where B is the number of blocks in a track. However, in order to account for the fact that these $\frac{R}{P}$ blocks span over track and cylinder boundaries, we add corresponding correction terms and obtain

$$E[t_{trans}(R, P)] = (n_{hs} - n_{cs})t_{hs} + n_{cs}t_{cs} + \frac{R}{PB} ROT \,, \tag{12}$$

where

- n_{hs} is the number of head switches (including cylinder switches),
- n_{cs} is the number of cylinder switches,
- t_{hs} is the head switch delay, and
- t_{cs} is the cylinder switch time (time for a seek of distance 1).

Using simple probability arguments we estimate

$$\begin{aligned} n_{hs} &= \lceil \frac{R/P}{B} \rceil - 1 + \frac{B - (R/P - (\lceil \frac{R/P}{B} \rceil - 1)B) - 1}{B} \\ &\approx \frac{\frac{R}{P} - 1}{B} \,, \end{aligned} \tag{13}$$

$$\begin{aligned} n_{cs} &= \lceil \frac{R/P}{TB} \rceil - 1 + \frac{TB - (R/P - (\lceil \frac{R/P}{TB} \rceil - 1)B) - 1}{TB} \\ &\approx \frac{\frac{R}{P} - 1}{TB} \,, \end{aligned} \tag{14}$$

and we obtain

$$\begin{aligned} E[t_{trans}(R, P)] = &(\frac{\frac{R}{P} - 1}{B} - \frac{\frac{R}{P} - 1}{TB})t_{hs} \\ &+ \frac{\frac{R}{P} - 1}{TB} t_{cs} + \frac{R}{PB} ROT \,, \end{aligned} \tag{15}$$

where T is the number of tracks in a cylinder.

Combining the above results we obtain the following formula for the expected service time:

$$\begin{aligned} E[T_{serv}(R, P)] = &\, eC(1 - a - bln(P)) + f + \frac{P}{P+1} ROT \\ &+ (\frac{\frac{R}{P} - 1}{B} - \frac{\frac{R}{P} - 1}{TB})t_{hs} + \frac{\frac{R}{P} - 1}{TB} t_{cs} \\ &+ \frac{R}{PB} ROT \,. \end{aligned} \tag{16}$$

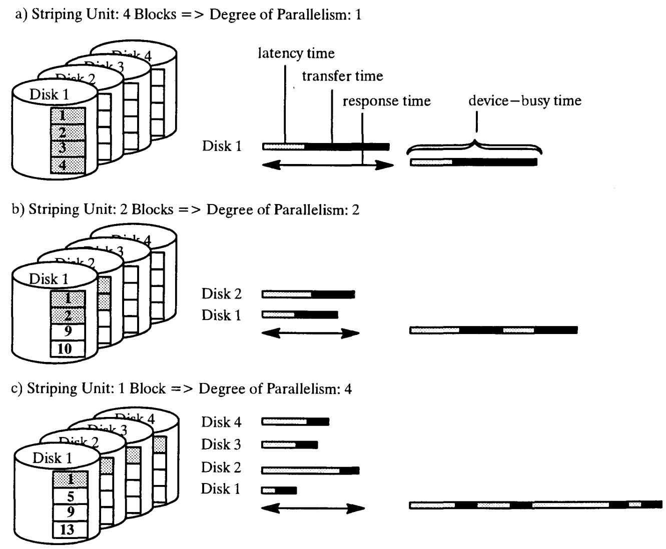

Fig. 1. Striping with different striping units

The trade-offs between increased seek and rotation time on one hand and reduced transfer time on the other hand for various degrees of parallelism are illustrated also in Fig. 1. This example considers a file that is being striped across four disks with three different striping units and resulting degrees of intra-request parallelism. The figure traces the execution of an I/O request of size four blocks for the three configurations. For illustration purposes, the seek and rotation times are combined together into latency time.

The optimal degree of parallelism, P_{opt}, can be determined by finding the minimum of the function $E[T_{serv}(R, P)]$, i.e., by solving the following cubic equation for P:

$$\begin{aligned} \frac{dE[T_{serv}(R, P)]}{dP} = &\frac{ROT}{P+1} - \frac{eCb}{P} - \frac{PROT}{(P+1)^2} \\ &+ (\frac{R}{P^2TB} - \frac{R}{P^2B})t_{hs} \\ &- \frac{R}{P^2TB}t_{cs} - \frac{RROT}{P^2B} \\ = &\, 0 \,. \end{aligned} \tag{17}$$

2.2 Phase B: minimizing response time by considering throughput and queueing delay

An increased degree of parallelism leads not only to trade-offs between seek and rotation time on one hand and reduced transfer time on the other hand, but also adversely affects the device-busy time of a request, i.e., the sum of the times that the disks are involved in the request and hence are not available for other requests. The relationship between the device-busy time and the various components of the response time is also illustrated in the execution of requests in Fig. 1. The throughput, measured as the number of requests completed per time unit, is inversely proportional to the average device-busy time of a request. Thus, higher degrees of parallelism lead to "unproductive" positioning times and, hence, to lower throughput.

The Phase A model for service time minimization has assumed that there are no interferences among the various requests and that no queueing delays occur. This is obviously not the situation in a multiprogramming environment; especially under heavy load, i.e., a high arrival rate, queueing delays play an important role. The scenario where each I/O request is served by a single disk is well understood and can be modeled via an M/G/1 queueing model [39]. We observe, however, that no general analytical model is known for so-called fork-join queueing model [52,62], i.e., for the case when I/O requests are served by multiple disks and the number of disks involved varies from request to request. An exception is the case when exactly two disks are being involved in serving every request [23,24].

We present in this section a simplified and computationally tractable analytic approximation to the fork-join model, under the assumption of perfect load balance. More specifically, we compute first the mean response time on each disk, assuming that the requests are equally distributed among the disks and that each disk can be represented as an M/G/1 system. Then, we use an approximation method outlined in [45] in order to compute the expected response time for requests with degree of parallelism $\overline{P}$ (averaged over the requests to all files) as the maximum among the response times of the $\overline{P}$ participating disks.

Our analytic approximation to the queueing model requires that we provide an estimate of the average arrival rate to all files in the system, denoted as $\overline{\lambda}$, in addition to the average request size, across all files, denoted as $\overline{R}$. Note that the value of $\overline{R}$ can be derived by sampling, or, alternatively, it can be computed from the average request sizes R_i to the individual files and the access frequencies of the files. The objective of Phase B is to compute the optimal value for $\overline{P}$, the average degree of parallelism.

Given that requests in our system have an average arrival rate of $\overline{\lambda}$ and an average degree of parallelism of $\overline{P}$, we obtain the overall arrival rate for the constituent subrequests as $\overline{\lambda} * \overline{P}$. Under the assumption of a perfectly balanced system where the subrequests are equally distributed among the disks, the subrequest arrival rate to a given disk i ($i = 1, \ldots, D$), to be denoted as λ_i, can be computed as

$$\lambda_i = \frac{\overline{\lambda}\,\overline{P}}{D}\,, \tag{18}$$

with D standing for the number of disks in the system.

The average subrequest size, to be denoted as $\overline{S}$, can be derived as

$$\overline{S} = \frac{\overline{R}}{\overline{P}}\,. \tag{19}$$

The service time for an individual subrequest to disk i, to be denoted by $t_{serv,i}(\overline{S})$, can be computed by using the standard formulae for the service time of a single disk. We can express the utilization of disk i, ρ_i, as

$$\rho_i = \lambda_i * t_{serv,i}(\overline{S})\,. \tag{20}$$

Using our assumption that each disk can be viewed as an M/G/1 queue, the expected value of $t_{resp,i}(\overline{S})$, the response time of the subrequests served at disk i, is given as [39]:

$$E[t_{resp,i}(\overline{S})] = E[t_{serv,i}(\overline{S})] + \rho_i * E[t_{serv,i}(\overline{S})] * \frac{1 + {c_i}^2}{2(1-\rho_i)}\,, \tag{21}$$

where c_i^2 stands for the squared coefficient of variation of the service time of subrequests at disk i. c_i^2 is defined as the ratio of the corresponding variance (VAR) and expected service time:

$$c_i^2 = \frac{VAR[t_{serv,i}(\overline{S})]}{E[t_{serv,i}(\overline{S})]^2}\,. \tag{22}$$

Also from M/G/1 queueing theory we obtain the formula below which relates the variance of the response time of individual subrequests on a disk i to the first three moments of their service time:

$$\mathit{VAR}[t_{resp,i}(\overline{S})] = \mathit{VAR}[t_{serv,i}(\overline{S})] + \frac{\lambda_i E[t_{serv,i}(\overline{S})^3]}{3(1-\rho_i)} + \frac{\lambda_i^2 E[t_{serv,i}(\overline{S})^2]^2}{4(1-\rho_i)^2}\,. \tag{23}$$

The response time for requests of size $\overline{R}$ served by $\overline{P}$ disks, to be denoted as $T_{resp}(\overline{R}, \overline{P})$, satisfies the equality

$$E[T_{resp}(\overline{R}, \overline{P})] = \max_i(t_{resp,i}(\overline{S}))\,. \tag{24}$$

In order to derive an analytic expression for the above equation, we make use of an approximation method presented in [45], which has been shown to be quite accurate if the response times of the individual subrequests, i.e., $t_{resp,i}$, obey a normal distribution. This approximation states that the expected response time for a request can be estimated as the response time of an individual subrequest plus a "correction" factor, which accounts for the slowest subrequest:

$$E[T_{resp}(\overline{R}, \overline{P})] \approx \begin{cases} E[t_{resp,i}(\overline{S})] + \sqrt{\mathit{VAR}[t_{resp,i}(\overline{S})]}\,\frac{\overline{P}-1}{\sqrt{2\overline{P}-1}} & \text{for } \overline{P} \le 3, \\ E[t_{resp,i}(\overline{S})] + \sqrt{\mathit{VAR}[t_{resp,i}(\overline{S})]}\sqrt{2\log \overline{P}} & \text{for } \overline{P} > 3. \end{cases} \tag{25}$$

We have conducted a series of experiments and these have shown that the assumption of normally distributed response times for the individual subrequests is a valid one.

We observe that with an increase in the variance of the response time of individual subrequests, the correction factor increases correspondingly. The impact of the degree of parallelism $\overline{P}$ on the different components of the response time we described informally in Fig. 1 is taken into account implicitly by the correction factor in Eq. 25.

In order to calculate $E[T_{resp}]$ in Eq. 25, it is necessary to compute the first three moments of the subrequests' service time distribution, namely, $E[t_{serv,i}(\overline{S})]$, $E[t_{serv,i}(\overline{S})^2]$, and $E[t_{serv,i}(\overline{S})^3]$. For this calculation, we need to derive the probability density function of $t_{serv,i}(\overline{S})$. The probability density functions of the corresponding seek and rotation times, i.e., $f_{seek,i}$ and $f_{rot,i}$, can be derived from Eqs. 5 and 9, respectively; the probability density function of the transfer time, i.e., $f_{trans,i}$, is a constant whose value is obtained by setting $P = 1$ in Eq. 12. Finally, the probability density function for $t_{serv,i}(\overline{S})$ can be obtained by convoluting the probability density functions $f_{seek,i}$, $f_{rot,i}$, and $f_{trans,i}$. The full details of this derivation are given in [84].

The value $\overline{P}$ which minimizes Eq. 25 can be found iteratively, by going through the range of possible values for $\overline{P}$ (this is obviously bounded by D, the number of disks in the system). We choose this approach, since Eq. 25 is not easily differentiable, unlike its counterpart in Phase A, namely, Eq. 16.

2.3 Putting it all together: the algorithm for data partitioning

The complete algorithm for data partitioning is outlined in Fig. 2 below. If we anticipate a low arrival rate of requests and desire to perform optimization only by using Phase A, then Steps 2 and 3 are omitted. For file-specific partitioning, Steps 1, 3, and 4 need to be iterated over the number of files in the system. On the other hand, for global partitioning, the above steps need to be executed only once, with one exception as explained below.

The effective degree of parallelism, P_{eff}, is computed in Step 3 by choosing the minimum between the degrees of parallelism computed in Steps 1 and 2. The factor $R_i/\overline{R}$ is used to normalize the outcome of Step 2. This is due to the fact that, for requests larger than $\overline{R}$, we want the degree of parallelism of file i to exceed $\overline{P}$, and if R_i is smaller than $\overline{R}$, then $P_{eff,i}$ should be smaller than $\overline{P}$.

The striping unit and striping width are then derived from $P_{eff,i}$ (or P_{eff}, respectively) in Step 4. If all I/O requests start at run boundaries, then the striping unit of a file, SU_i, can be derived by using the formula $\lceil \frac{R_i}{P_{eff,i}} \rceil$. This is also the case when the requests are for individual blocks, i.e., R_i = 1, or for the entire file, i.e., $R_i = L_i$, with L_i being the file size. On the other hand, if the requests can start at any block inside a run, the formula above yields a striping unit which cannot support in most cases the degree of parallelism $P_{eff,i}$; this in fact increases $P_{eff,i}$ by one. In order to cover this case, the striping unit is derived by the alternative formula $\lceil \frac{R_i-1}{P_{eff,i}-1} \rceil$, which guarantees a degree of parallelism of $P_{eff,i}$ in all cases. In the case of $P_{eff,i}$ = 1, the striping unit should be chosen as large as possible, i.e., $SU_i = L_i$, with L_i being the file size. Finally, the striping width, denoted as SW_i, is chosen as high as possible in order to also support inter-request parallelism, in addition to the intra-request parallelism considered by the above steps. Notice that the striping width SW_i needs to be computed individually, also in the case of global partitioning, since some files may be too small to be spread over all the disks.

The algorithm outlined in Fig. 2 accomplishes static partitioning, since all the files are allocated at the same time. However, the algorithm can be extended easily to perform dynamic partitioning. Dynamic partitioning and the complementary procedure of incremental repartitioning need to be performed when new files are added, old files are deleted, or when the access characteristics of some files change substantially. Let us discuss here the case when a new file needs to be added to the system. We need to recompute first the access characteristics specified in the input to the partitioning algorithm, i.e., to readjust $\overline{R}$, $\overline{S}$, and $\overline{\lambda}$ in order to account for the addition of the new file. In order to perform these calculations we need to estimate R_i, the average file request size, as well as λ_i, the average arrival rate of requests to the new file. As discussed before, this information may be derived by sampling existing files of the same type, or may be provided as a hint by the database administrator (e.g., when

Input:
- D = number of disks
- $\overline{R}$ = average request size over all files
- R_i = average request size of file i – for file-specific partitioning
- $\overline{L}$ = average file size
- L_i = size of file i – for file-specific partitioning
- $\overline{\lambda}$ = average arrival rate of requests

Output:
- SU_i = near-optimal striping unit of file i – for file-specific partitioning
- SU = near-optimal global striping unit
- SW_i = near-optimal striping width of file i

Step 1: Apply Phase A optimization with respect to service time
- a. File specific partitioning:
  determine $P_{opt,i}$, the optimal degree of parallelism for file i, by setting $R = R_i$ in equation (16)
- b. Global partitioning:
  determine P_{opt}, the approximately optimal average degree of parallelism, by setting $R = \overline{R}$ in equation (16)

Step 2 : Apply Phase B optimization to determine $\overline{P}$, the approximately optimal (average) degree of parallelism for the requested throughput, $\overline{\lambda}$.

Step 3: Determine the effective degree of parallelism
- a. File specific partitioning:
  $P_{eff,i} = \min_i(P_{opt,i}, \overline{P}R_i/\overline{R})$
- b. Global partitioning:
  $P_{eff} = \min(P_{opt}, \overline{P})$

Step 4: Determine near-optimal striping unit and width
- a. File specific partitioning:
  $SU_i = \lceil (R_i-1)/(P_{eff,i}-1) \rceil$ for $1 < R_i < L_i$
  $\lceil R_i/P_{eff,i} \rceil$ otherwise
  $SW_i = \min_i(D, \lceil L_i/SU_i \rceil)$
- b. Global partitioning:
  Compute SU as in step 4a by replacing R_i by $\overline{R}$, L_i by $\overline{L}$, and $P_{eff,i}$ by P_{eff}
  Compute SW_i as in step 4a

Fig. 2. Data-partitioning algorithm

we consider a large application). We then invoke the static partitioning algorithm given above on the new file in order to determine its effective degree of parallelism, $P_{eff,i}$, and its striping unit and width.

A companion incremental repartitioning procedure is invoked periodically. This procedure checks first if a trigger condition is satisfied in order to warrant incremental repartitioning. The trigger condition consists of two parts:

(1) $\overline{P}_{new} \neq \overline{P}_{old}$ and
(2) $E[t_{resp}(\overline{R}_{new}, \overline{P}_{new})] < E[t_{resp}(\overline{R}_{old}, \overline{P}_{old})] - \epsilon$,

with ϵ being a system determined hysteresis parameter. The new set of statistics is computed by performing Steps 1 through 3 of the static partitioning algorithm; the old set of statistics is the one computed at the last invocation of this procedure. If the trigger condition is satisfied, then we proceed to do incremental repartitioning of a fixed number of files. The procedure considers candidate files for reorganization by using a list in which the files are sorted in descending order of heat. We use heat as an ordering criterion since this measures the product of arrival rate and file size; an early repartitioning of the hottest files will make the biggest contribution to the average degree of parallelism P_{opt}. Note that, although $\overline{P}_{new}$ may be different from $\overline{P}_{old}$, a particular file i may not need to be reorganized if the value of $P_{eff,i}$ does not change. The number of files to be considered for reorganization during one period is a system parameter that is chosen in advance.

3 Load balancing

The need for load balancing was already mentioned in Sect. 1 in the context of data allocation. Recall that load balancing does not become obsolete when striping is employed. Many applications require that we choose large striping units in order to achieve a certain throughput with multi-block requests. For example, Gray et al. have proposed the parity striping scheme [33], where the distribution of data blocks is based on a very large (possibly infinite) striping unit, and similar results on the throughput limits of fine-grained striping have been stated in [12, 13, 48, 51, 59, 79, 80]. However, a coarser striping unit increases the probability of load imbalance under a skewed workload [13, 51]. Addressing this tradeoff solely by tuning the striping unit is only a (bad) compromise. Thus, additional methods for load balancing are called for, regardless of whether data is partitioned or not.

Obviously, the load balance of a disk system depends on the placement of data, regardless of whether the files are partitioned or not. The data placement problem is similar to the file allocation problem in distributed systems [19] and falls in the class of NP-hard problems (the simplest case is equivalent to the NP-complete problem of multiprocessor scheduling; see problem [SS8] in [26]). Hence, viable solutions must be based on heuristics. The worst case performance of these heuristic methods can be measured in terms of their competitive ratio, which is defined as the ratio between the heat of the hottest disk under a given heuristic placement and the heat of the hottest disk under an optimal placement. Good heuristics based on greedy placement [32] or iterated bin-packing [17] are well understood for the static file allocation problem with non-partitioned files, where the heat of each file is known in advance. In the greedy algorithm,which was adopted in the Bubba parallel database machine [14], the files are first sorted by descending heat and then they are allocated in this order, where in each step the disk with the lowest accumulated heat is selected. Under this greedy heuristic, the competitive ratio is bounded by $\frac{4}{3} - \frac{1}{3 \times \text{number of disks}} < 1.34$, while for the iterated bin-packing algorithm of [17] the corresponding competitive ratio is approximately 1.22. We observe here that these results are derived from specifically constructed "adversary" inputs, and there is experimental evidence that these heuristic allocation algorithms perform better for most realistic inputs.

In practice, realistic algorithms for static allocation of non-partitioned files need to consider additional parameters and system constraints such as controller contention and storage space limitation. A comprehensive, heuristic optimization method which considers some of these constrained is presented in [82], where a non-linear programming solution embedded in a queueing network model is described. Moreover, in many application environments, the files are not allocated all at the same time, but rather some files are allocated dynamically. For this dynamic case, the following "canonical" extension of the greedy heuristic mentioned above has been studied intensively in the theory of online algorithms: a new file is placed on the disk with the currently lowest accumulated heat, and the heat of the target disk is then incremented by the heat of the new file. It has been shown that this online greedy method guarantees a competitive ratio of $r = 2 - \frac{1}{\text{number of disks}}$ [32]; This worst case bound can be further improved, to a minor extent, by more sophisticated allocation heuristics [5, 40]. However, it has also been shown that no online algorithm can achieve a competitive ratio better than $1 + \frac{1}{\sqrt{2}} \approx 1.7$ [22]. When additional constraints on the set of eligible disks are taken into account, the best possible competitive ratio is bounded (from below) by $1 + \lceil \log_2(\text{number of disks}) \rceil$ [3]. The problem of data allocation in parallel disk systems has an additional constraint that is not considered in any of the works mentioned above. Namely, in order to support intra-request parallelism, it is necessary to allocate the extents of a file on different disks.

Not only are files to be created or deleted dynamically, but files can grow or shrink. In addition, the access characteristics of files can change over time, and what was originally a good allocation under a certain workload may not be any longer the case later in time. In order to deal with all these dynamics of change, it is necessary to incorporate into a file manager another tuning component that can redistribute the load by migrating data from one disk to another at any time a certain imbalance in load is detected. Migration of entire files has been considered in the context of replicated file systems. On the other hand, migration of file portions has been considered for scalable, distributed hashing schemes but with different objective functions [2, 6, 9, 50, 75, 76, 83]. The only work that considers data migration in the context of disk load balancing is [38]; however, this work is restricted to offline and monolithic (i.e., non-incremental) reorganization.

The load-balancing component of our intelligent file system consists of two independent modules: one that performs file allocation and a second one that performs dynamic redistribution of data. These components are described in Subsects. 3.1 and 3.2. Subsect. 3.3 explains how our system keeps track of the heat and temperature of extents and disks.

3.1 Data allocation

We have extended the greedy algorithm of [32] in order to deal with (dynamic) allocation of partitioned files [79]. In the static case where all files are given in advance, the algorithm first sorts all extents by descending heat and the extents are allocated in sort order. For each extent to be allocated, the algorithm selects the disk with the lowest accumulated heat among the disks which have not yet been assigned another extent of the same file. This method is illustrated in Fig. 3 and is contrasted with a standard round-robin scheme. The figure shows the placement of three files each consisting of three extents with heat proportional to the height of the corresponding boxes. We denote by i.j the extent j of file i. Observe that in Fig. 3 extents 2.2, 1.2, and 3.1 are allocated

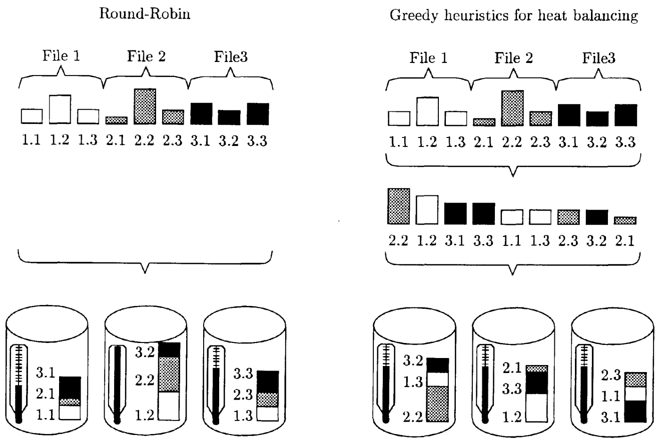

Fig. 3. Illustration of static allocation heuristics

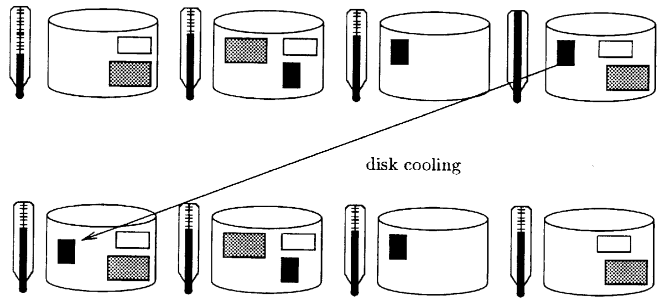

Fig. 4. Illustration of "disk cooling"

in this order to the current disk with the lowest accumulated heat; however, when extent 3.3 is to be allocated, we do not choose disk 3, since it holds already an extent of file 3, but instead of this allocate it on disk 2.

In the dynamic case, the sorting step is eliminated and the algorithm uses only the information about the heat of the files which have been allocated and for which statistics are collected already. Thus, as compared to the canonical extension discussed in the previous section, the heat of the target disk remains unchanged at the time of an extent allocation. The heat will be adjusted correspondingly only after enough accesses to the newly allocated extent have been recorded.

The disk selection can be made in such a way as to consider also, if so desired, the cost of additional I/Os necessary to perform partial disk reorganization. Partial disk reorganization may have to be performed if, due to file additions and deletions, there is room to store an extent on a disk but the space is not contiguous. Even more expensive is the situation when disk i has the lowest heat and may appear as the obvious choice to store a new extent of a file, but this disk does not have enough free space. In order to make room for the new extent, we have to migrate one or more extents to a different disk. In order to account for these reorganization costs, we associate with every disk a status variable with regard to the extent chosen for allocation. The status variable can take the values `FREE`, `FRAG`, and `FULL`, depending upon whether the disk (1) has enough free space for the extent, (2) has enough space but the the space is fragmented, or (3) does not have enough free space. Our file allocation algorithm has the option of selecting disks in increasing heat order without regard to their status. Alternatively, we may select the disks in multiple passes, where in the first pass we only choose those that have status `FREE`. More details and experimental studies on this combined free-space management and data allocation method are given in [79]. In the current paper, we do not further consider the impact of fragmented or full disks.

3.2 Disk cooling

In order to perform dynamic heat redistribution, we employ in our system a dynamic load-balancing step, called disk cooling. Basically, disk cooling is a greedy procedure which tries to determine the best candidate, i.e., extent, to remove from the hottest disk in order to minimize the amount of data that is moved, while obtaining the maximal gain. The temperature metric is used as the criterion for selecting the extents to be reallocated, because temperature reflects the benefit/cost ratio of the reallocation, since benefit is proportional to heat (i.e., reduction of heat) and cost is proportional to size (of the reallocated extents). This approach is illustrated in Fig. 4; the basic disk-cooling algorithm is given in Fig. 5. The extent to be moved, denoted by e, is reallocated on the coolest disk, denoted by t, such that t does not hold already an extent of the corresponding file and t has enough contiguous free space.

In our system the disk-cooling procedure is implemented as a background demon which is invoked at fixed intervals in time. The procedure checks first if the trigger condition is satisfied or not (Steps 1 and 2 in Fig. 5). If the trigger condition is false, the system is considered load-balanced and no cooling action is performed. In the basic disk-cooling procedure, the system is not considered load balanced if the heat of the hottest disk exceeds the average disk heat by a certain quantity δ. It is important to observe that during each invocation of the procedure different disks can be selected as candidates for cooling after each cooling step.

Our procedure considers implicitly the cost/benefit ratio of a considered cooling action and only schedules it for execution if it is considered beneficial. These cost considerations are reflected in Step 5 of the algorithm. The hottest disk is likely to already have a heavy share of the load, which we can "measure" by observing if its queue is non-empty. A cooling action would most likely increase the load imbalance if a queue is present at the source disk, since it implies additional I/Os for the reorganization process. Hence, we choose not to schedule the cooling action if this condition is satisfied. We also consider the cooling move not to be cost-beneficial if the heat of the target disk after such a potential move would exceed the heat of the source disk. Hence, although our background demon is invoked a fixed number of times, only a fraction of these invocations result in data migration.

Our generic disk-cooling procedure can be generalized in a number of ways. In [72], we have shown how an explicit objective function based on disk heat variance (DHV) can be used in a more general test for the cost/benefit of a cooling action. Thus, the benefit is computed by comparing the DHV after the potential cooling step with the DHV before the potential cooling step. In addition, we can also consider

Input: D - number of disks
H_j - heat of extent j
$H_i^\star$ - heat of disk i
$\overline{H}$ - average disk heat
E_i - list of extents on disk i sorted in descending temperature order
$\overline{D}$ - list of disks sorted in ascending heat order

Step 0: Initialization: target = not_found
Step 1: Select the hottest disk s
Step 2: Check trigger condition:
    `if` $H_s > \overline{H} \times (1 + \delta)$ `then`
Step 3:   `while` (E_s not exhausted) and (target == not_found) `do`
        Select next extent e in E_s
Step 4:   `while` ($\overline{D}$ not exhausted) and (target == not_found) `do`
            Select next disk t in $\overline{D}$ in ascending heat order
            `if` (t does not hold an extent of the file to which e belongs)
              `and` STATUS(t) == `FREE then`
              target = found
            `fi`
        `endwhile`
    `endwhile`
Step 5 :   `if` s has no queue `then`
        $H_s^{\star\prime} = H_s^\star - H_e$
        $H_t^{\star\prime} = H_t^\star + H_e$
        `if` $H_t^{\star\prime} < H_e$ `then`
            reallocate extent e from disk s to disk t
            update heat of disks s and t:
            $H_s^\star = H_s^{\star\prime}$
            $H_t^\star = H_t^{\star\prime}$
        `fi`
    `fi`
`fi`

Fig. 5. Basic disk cooling algorithm

explicitly the cost of performing the cooling. Thus, a more accurate calculation of benefit and cost would consider not only the reduction in heat on the origin disk and the increase in heat on the target disk, but also the additional heat caused by the reorganization process itself. The cooling process is executed during two intervals of time, the first corresponding to the read phase of the action and the second corresponding to the write phase of the action. The additional heat generated during these phases can be computed by dividing the size of the extent to be moved by the corresponding duration of the phase. The duration times of the read and write phase of a cooling action can be estimated by using a queueing model, as shown in [72].

Our disk-cooling procedure can be fine-tuned so that the unit of reallocation is chosen dynamically in order to increase the potential of a positive cost/benefit ratio. In the basic procedure given in Fig. 5, the unit of redistribution is assumed to be an extent. However, in the case of large extents that are very hot, the cost of a redistribution may be prohibitive. In this case, we can subdivide further an extent into a number of fixed-size *fragments* and use a fragment as the unit of redistribution. Since all fragments of an extent are of the same size, we can now base the choice of the migration candidates (see Step 3 in Fig. 5) on the heat statistic instead of temperature. Note that an increase in the number of allocation units of a file also requires that we remove the allocation constraint on the target disk, namely we do not require anymore that the disk should hold only one fragment of a file. Hence, we put here the objective of a balanced load above the requirement that the file partitioning is optimal.

3.3 Heat tracking

The dynamic tracking of the heat of blocks is implemented based on a moving average of the inter-arrival time of requests to the same block. Conceptually, we keep track of the times when the last k requests to each block occurred, where k is a fine-tuning parameter (in the range from 5 to 50). To illustrate this bookkeeping procedure, assume that a block is accessed at the points in time $t_1, t_2, \ldots, t_n$ $(n > k)$. Then the average inter-arrival time of the k last requests is $\frac{t_n - t_{n-k+1}}{k}$, and the estimated heat of the block is the corresponding reciprocal $\frac{k}{t_n - t_{n-k+1}}$. Upon the next access to this block, say at time t_{n+1}, the block heat is re-estimated as $\frac{k}{t_{n+1} - t_{n-k+2}}$.

One may conceive an alternative method for heat tracking that keeps a count of the number of requests to a block within the last T seconds, where T would be a global tuning parameter. The problem with such a global approach is that it cannot track the heat of both hot and cold blocks in an equally responsive manner. Hot blocks would need a relatively short value of T to ensure that we become aware of heat variations quickly enough. Cold blocks, on the other hand, would need a large value of T to ensure that we see a sufficient number of requests to smooth out stochastic fluctuations. The moving-average method for the inter-arrival time does not have this problem, since a fixed value of k actually implies a short observation time window for hot blocks and a long window for cold blocks. Moreover, extensive experimentation with traces from real applications with evolving access patterns has shown that our tracking method works well for a wide spectrum of k values; the heat estimation is fairly insensitive to the exact choice of k [84]. Furthermore, under the assumption that requests to a block arrive according to a Poisson process (i.e., with exponentially distributed inter-arrival time), the heat estimate would be Erlang-k distributed and the minimum k for achieving a desired statistical confidence in the heat estimate can be derived analytically [46].

The adopted heat-tracking method is very responsive to sudden increases in a block's heat; the new access frequency is fully reflected in the heat estimate after k requests, which would take only a short while for hot blocks (and reasonable values of k). However, the method adapts the heat estimate more slowly when a block exhibits a sudden drop of its heat. In the extreme case, a hot block may suddenly cease to be accessed at all. In this case, we would continue to keep the block's old heat estimate, as there are no more new requests to the block. To counteract this form of erroneous heat estimation, we employ an additional "aging" method for the heat estimates. The aging is implemented by periodically invoking a demon process that simulates "pseudo-requests" to all blocks. Whenever such a pseudo-request would lead to a heat reduction, the block's heat estimate is updated; otherwise the pseudo-request is ignored. For example, assume

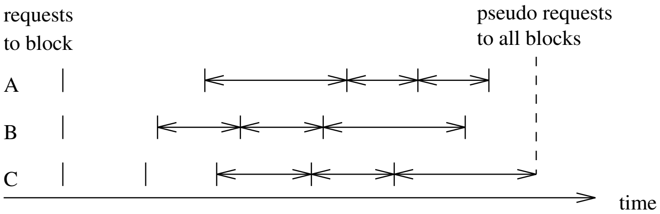

Fig. 6. Illustration of the heat tracking method for k = 3. The relevant interarrival times are shown by the double-ended arrows

that there is a pseudo-request at time t' and consider a block with heat H. We compute tentatively the new heat of the block as $H' = \frac{k}{t' - t_{n-k+2}}$, but we update the heat bookkeeping only if $H' < H$. The complete heat-tracking method is illustrated in Fig. 6.

The described heat-tracking method requires a space overhead of $(k+1)$ floating-point numbers per block. Since we want to keep this bookkeeping information in memory for fast cooling decisions, it is usually unacceptable to track the heat of each individual block. In order to reduce the overhead involved in heat tracking, we actually apply the heat estimation procedure to entire extents (or fragments of a specified size). We keep track of the times $t_n, \ldots, t_{n-k+1}$ of the last k requests that involve any blocks of the extent in the manner described above, and we also keep the number of accessed blocks within the extent for each of the last k requests. Assume that the average number of accessed blocks is R. Then the heat of the extent is estimated by $\frac{kR}{t_n - t_{n-k+1}}$. Finally, we estimate the heat of a fraction of an extent by assuming that each block in the extent has the same heat (which is extent heat divided by extent size). This extent-based heat-tracking method reduces substantially the space overhead of the block-based estimation procedure.[1] On the other hand, our experimental studies (including studies with application traces) have shown that the loss in accuracy versus block-based heat tracking is minimal.

4 Experimental results

In this section, we present an experimental performance evaluation of the file striping and allocation and load-balancing algorithms presented above. The testbed for these experiments was built on top of the file system prototype FIVE [84]. FIVE runs on shared-memory multiprocessors under Solaris and a few other Unix versions. It can manage either real data on real disks (i.e., raw partitions), or it can interact with a simulated disk system to estimate the impact the virtual resources. The disk simulation keeps track of exact arm positions as well as rotational positions of the disk head. Our simulator considers head switch delays and incorporates a realistic estimation of the seek time as a nonlinear function of the seek distance, as well as other details of real disks [67]. In the simulation mode, FIVE makes use of the process-oriented simulation library CSIM [73], which manages the bookkeeping for the virtual disks (e.g., disk queues). For the experiments reported here we used a simulated parallel disk system, whose parameters are described in Table 1.

[1] Additional approximation techniques to further decrease the space overhead are described in [84]. When memory consumption is extremely critical, one can even employ an approximation that requires only keeping the values of t_n and t_{n-k+1} and thus has constant space overhead independently of k.

Table 1. Hardware characteristics of the simulated disk system

# disks	32	capacity of one disk	539 MB
block size	1 KB	capacity of the disk system	17.2 GB
track size	35 blocks	revolutions per minute	4400 rpm
# tracks per cylinder	11	average seek time	12 ms
# cylinders per disk	1435	transfer rate per disk	2.44 MB/s

FIVE allows for the striping of files on an individual or global basis and incorporates heuristic algorithms for file striping, allocation, and dynamic load balancing, as described in Sects. 2 and 3. These algorithms can be invoked online, i.e., concurrently with regular requests to existing files. We have implemented a load generator that can generate synthetic workloads according to specified parameter distributions, or analyze (and filter) existing traces and feed them as input to FIVE. For the performance studies reported here, we mostly relied on synthetic workloads, for which we could control and systematically vary all relevant parameters. A representative set of experiments with a synthetic workload is described in Subsect. 4.1. We also report on disk-cooling studies using two trace-based experiments in Subsect. 4.2. Further trace-based performance studies with FIVE can be found in [84].

4.1 Experiments with synthetic workload

For these experiments, we generated a set of 10,000 files and two types of workloads, one with a uniform access pattern and the second with a skewed access pattern, as we shall describe in more detail below. The files themselves were identical for both workloads, and in both cases each (read or write) request accessed an entire file. The file sizes were hyperexponentially distributed such that each file belongs to one of three different classes with certain mean values (and exponential distribution of file sizes within each class). Files of class A had a mean size of 20 KB, files of class B had a mean size of 500 KB, and files of class C had a mean size of 1000 KB. Class-C files were not accessed in the generated workload; they represent "passive" data that occupy disk space and thus influence seek times. Class-A files represent relatively small data objects, e.g., simple HTML documents on the WWW. Class-B files, on the other hand, represent relatively large multimedia data objects. The important point here is that the workload covered a wide spectrum of request sizes, which we consider to be a particular challenge of advanced applications such as HTTP servers, multimedia information systems, and object-oriented database systems. In both workloads, we assigned the same probability of selection to files from the two classes A and B. Table 2 summarizes the common characteristics of both synthetic workloads.

4.1.1 Workload with uniform access frequencies

In this subsection, we consider a workload with uniform access frequencies: read and write accesses are generated

Table 2. Characteristics of the synthetic workload

# of files of class A	1000
# of files of class B	1000
fraction of files of class A	0.5
fraction of files of class B	0.5
average size of file class A	20 KB
average size of file class B	500 KB
overall average request size	260 KB
standard deviation of request size	416 KB
read fraction	0.7

to file classes of type A or B, such that each file within a class has the same probability of selection. We generated a sequence of 1,000,000 file requests with exponentially distributed inter-arrival times.

We compared first the response time of five different striping strategies, namely a file-specific one (*Opt*) and four global strategies (*Gopt, Block, Track, Cylinder*) under light load (i.e., an arrival rate of 1 request/second), so that queueing effects were negligible. These striping strategies are

1. *Opt*: files are partitioned based on the first step of the heuristic approach described in Sect. 2 that minimizes response time in single-user mode.
2. *Gopt*: the striping unit for each file is set to 8 KB, which was determined by using the first step of the heuristic method of Sect. 2 under the assumption that all files have the same average request size $R = 260$ KB.
3. *Gbest*: the striping unit for each file is the best global value (5 KB) that we found by exhaustive search of all possible values (i.e., this striping unit yielded the best response time averaged over all requests).
4. *Block*: the striping unit for each file is a block (i.e., 1 KB).
5. *Track*: the striping unit for each file is a track (i.e., 35 KB).
6. *Cyl*: the striping unit for each file is a cylinder (i.e., 385 KB).

Note that, for this first set of experiments, we assumed a light load, hence the striping unit for Opt and Gopt was computed without regard to throughput and queueing delay considerations. Table 3 shows the average response time for the five different striping methods; these performance figures are further broken down into different categories of request sizes in Table 4.

The Opt method outperforms all other methods, except Block striping (and the "hand-tuned" Gbest), for almost all request size categories, although the improvements over Gopt are minor. Block striping is competitive and even slightly superior for request sizes in the 10–100-KB range. Even for larger requests, the latency of the "slowest" disk rapidly approaches the maximum latency under both Opt and Block, so that the aggressive intra-request parallelism of the Block method does not incur an additional penalty once the degree of parallelism exceeds a certain number. However, as we will see below, the Block method exhibits severe drawbacks when the request arrival rate is increased so that disk arm contention and the resulting queueing delays become a factor, whereas the Opt method scales much better with increasing load.

Compared to the Track and Cyl methods, Opt achieves significant improvements in the order of 30% (in the case of

Table 3. Average response time in milliseconds of the synthetic workload under light load ($\lambda = 1$)

Opt	Gopt (8 KB)	Gbest (5 KB)	Block	Track	Cyl
24.54	24.75	24.21	24.54	28.86	81.84

Track) for medium to large requests between 50 and 500 KB. For very large requests, all methods (except Cyl) spread a file across all 32 disks, so that the performance differences eventually become negligible when the request size is further increased beyond 1 MB.

The global striping method Gopt turned out to be very competitive to the file-specific striping method Opt; the advantage of Opt is more or less negligible throughout the spectrum of request sizes. We also compared these two methods with Gbest, the best possible global striping strategy whose striping unit was found through exhaustive trials. For this particular workload, the Gbest method has a striping unit of 5 KB and its performance was almost identical to that of Opt and Gopt. So, although file-specific striping did not prove to be truly superior to global striping in these experiments, the positive conclusion from these light-load experiments is that our heuristic optimization method did indeed approximate the real optimum very well.

In order to take into account throughput requirements and queueing delays, we performed a second set of experiments, in which we varied the request arrival rate. For this set of experiments, we also considered two additional striping strategies, namely:

1. *Opt-140*: the file-specific striping unit is computed with the additional constraint that a request arrival rate of $\lambda = 140$ must be supported. Accordingly, files are partitioned subject to the constraint that the average degree of intra-request parallelism is bounded by 3 (as computed by the heuristics described in Sect. 2).
2. *Gopt-140*: the global striping unit is computed for a request arrival rate of $\lambda = 140$. The corresponding striping unit size is $\lceil 260/3 \rceil = 87$ KB.

Furthermore, we replaced the *Gbest* method by the best striping unit for the given request arrival rate of $\lambda = 140$, denoted as *Gbest-140*, which was again found through exhaustive search among all possible values and turned out to be 70 KB.

Table 5 shows the average response times of the various striping methods as a function of the request arrival rate, which was varied from 20 up to 140 requests per second. Note that although the figures show explicitly only response time, a fast-growing curve for response time implies that beyond a relatively small value for the arrival rate the throughput reaches saturation. This also explains the ∞ entries in Table 5: they denote those experiments where the arrival rate exceeded the sustainable throughput and thus led to excessive queueing and a continuously growing backlog of requests.

As Table 5 shows, the Opt method scales up with increasing arrival rate much better than Block striping. However, for sufficiently high arrival rate, Opt is clearly outperformed by Track and Cyl striping, the reason being that the latter two methods employ lower degrees of intra-request parallelism and can thus sustain higher load. The figures also

Table 4. Average response time in milliseconds for different request sizes of the synthetic workload under light load (λ = 1)

Request size [KB]	Opt	Gopt (8 KB)	Gbest (5 KB)	Block	Track	Cyl
$\leq$ 10	16.97	17.06	17.05	19.18	16.84	17.63
11–50	22.04	21.26	20.77	21.22	24.50	26.18
51–100	22.97	23.81	22.73	22.28	31.23	48.94
101–200	24.58	25.91	24.71	24.42	34.56	86.41
201–500	27.48	28.45	27.69	27.35	36.38	168.39
501–1000	33.83	34.66	34.05	33.79	38.46	200.38
$>$ 1000	49.75	50.01	49.63	49.65	53.29	207.27

Table 5. Average response time in milliseconds for the synthetic workload as a function of the request arrival rate λ

λ	Opt	Gopt (8 KB)	Block	Track	Cyl	Opt140	Gopt140 (87 KB)	Gbest140 (70 KB)
20	29.62	29.71	31.84	32.42	90.36	42.08	44.66	39.73
40	38.13	37.79	48.76	37.15	100.53	47.31	50.04	44.33
60	55.17	53.35	126.20	43.66	112.76	53.83	56.81	50.14
80	112.48	97.02	∞	53.16	128.00	62.31	65.59	57.66
100	∞	1033.67	∞	69.13	147.39	74.18	77.73	68.04
120	∞	∞	∞	102.86	173.50	91.64	96.21	83.61
140	∞	∞	∞	206.71	211.29	121.26	127.49	110.04

show the trend that Cyl will eventually pass Track, as it is even more conservative in terms of parallelism and resource consumption.

The striping methods that are specifically tuned for a particular arrival rate outperform both Track and Cyl by almost a factor of two (in the case of λ = 140). This demonstrates very nicely the need for application-specific tuning of striping units. We also determined through exhaustive trials the best possible striping units for the Gbest method under different arrival rates. For λ = 140 the response time of Gbest was approximately 110 ms, and this was obtained for a global striping unit of 70 KB. We note, however, that such a tuning method that is based on exhaustive trials is completely unfeasible in practice. Thus, the fact that both the Opt and the Gopt methods approached the real optima within approximately 10–15% is indeed a successful result and demonstrates the viability of our tuning heuristics.

In the above experiment, the Opt methods achieved only very small improvements over the corresponding Gopt methods. This almost negligible advantage of Opt over Gopt does not seem to justify the increased software complexity of file-specific striping. However, file-specific striping allows for incremental restriping of individual files when changing workload characteristics require higher I/O rates or data rates for some crucial files. A global striping unit strategy does not support this type of reconfiguration. Thus, for global striping, a change of the striping unit requires unloading all files, re-initializing the disk system with the new striping unit and reloading the data. This costly procedure leads to a significant downtime of the system. For this reason, we still believe that file-specific striping is an essential requirement for data management in parallel disk systems.

4.1.2 Workload with skewed access frequencies

In order to study the influence of data access skew and the effectiveness of our "disk-cooling" procedure, we have modified the synthetic workload of the previous subsection so that the distribution of file access frequencies followed a Zipf-like curve (everything else was identical to the previous setup). Thus, if the files are numbered from 1 to N, the probability of accessing a file numbered i, with i $<$ N, is given by the formula [47]:

$$Prob[i \leq s] = \left(\frac{s}{N}\right)^{\log(X/100)/\log(Y/100)}, \tag{26}$$

where X and Y are parameters that were set to 70 and 30, respectively. The parameter N denotes the number of active files, i.e., files in classes A and B, which is set to 2000 in our experiments. This probability distribution results in a self-similar, skewed access pattern, where a fraction X of the requests refers to a fraction Y of the files, and this skew is recursively repeated within the fraction X of "hot" files. Such skew patterns are common in many OLTP and database applications, and they have been observed for WWW servers as well [7].

In order to study the effects of load balancing in isolation, we did not perform any caching of the data in these experiments. Note that load balancing is still a crucial problem, even if caching is used. Caching would keep the hottest blocks in main memory, but the remaining blocks can still exhibit a significant access skew.

Table 6 shows the average response time results for this experiment as a function of the arrival rate λ. We considered three different striping strategies, namely, Gopt140, Track, and Cyl. We do not show explicitly the results for the strategies Block and Opt140; the Block strategy could sustain only a throughput of about 60 requests per second and started thrashing at this point, while the performance of the Opt140 strategy was almost identical to that of Gopt140. All files were pre-allocated based on a round-robin scheme, and we compared the case without cooling against the case with cooling switched on. The latter case is denoted by the "-C" suffix in Table 6. A cooling step was attempted every $100/\lambda$ seconds (i.e., equivalently, every 100 regular requests), the migration units were entire extents, and the load imbalance threshold δ was set to 5% (see Sect. 3.2).

Table 6. Average response time in milliseconds for the skewed synthetic workload as a function of the request arrival rate λ

λ	Gopt140 (87 KB)	Track	Cyl	Gopt140-C	Track-C	Cyl-C
20	46.57	33.38	98.34	47.23	34.23	96.66
40	53.30	38.92	124.66	53.59	39.98	113.91
60	62.81	47.05	∞	64.02	48.09	157.58
80	77.93	60.39	∞	77.73	61.68	∞
100	111.87	87.52	∞	110.63	87.20	∞
110	176.13	117.80	∞	117.20	109.72	∞
120	∞	203.30	∞	152.74	163.72	∞
125	∞	438.65	∞	188.88	221.42	∞
130	∞	∞	∞	199.63	429.76	∞

Table 7. Average response time in milliseconds for the dynamically evolving, skewed, synthetic workload as a function of the request arrival rate λ

λ	Gopt140 (87 KB)	Gopt140-C (87 KB)
20	44.75	45.15
40	50.39	50.83
60	57.95	58.37
80	69.11	68.97
100	90.91	85.83
110	120.68	101.65
120	921.66	133.50
130	∞	429.79

The response time figures demonstrate that access skew does have a disastrous effect on performance, unless it is counteracted by load balancing. For example, at an arrival rate of 120 requests per second, the average response time of Track striping without cooling degrades by a factor of 2 compared to the workload with uniform access frequencies. The underlying reason is that under the skewed load the hottest disk had a much higher utilization (and corresponding average queue length) than the overall disk system and thus formed a premature bottleneck; at λ = 120 the hottest disk had a utilization of 0.94 and an average queue length of 10.3, while the average disk utilization and queue length (averaged over all disks) were 0.79 and 3.5, respectively (under Track striping). The cooling procedure was able to reduce the utilization and the average queue length of the hottest disk down to 0.89 and 5.5, respectively, at λ = 120 and could thus improve the average response time significantly.

All methods without cooling started thrashing at an arrival rate of 130 requests per second or earlier. When approaching the thrashing region, the response times of any striping strategy with cooling switched on are an order of magnitude lower compared with the same strategy with cooling turned off. Note that cooling does incur a certain overhead by migrating extents between disks. This leads to a small increase of the overall disk utilization, and this is why the cooling methods exhibit a slightly higher response time than the no-cooling methods under light load. However, when the extra load due to cooling becomes a critical factor, cooling is deactivated automatically, as described in Sect. 3.2. An analysis of the invocation frequency distribution of cooling steps over the duration of an experiment shows that the cooling frequency is high in the first tenth of the experiment, and, as soon as the load is sufficiently balanced (as estimated by the heat bookkeeping), cooling is invoked only very infrequently due to occasional load fluctuations that exceed the imbalance threshold.

Among the three cooling variants that are shown in Table 6 the Gopt-λ-C method showed significant advantages over Track-C striping under high load, with response time improvements up to a factor of 2. This demonstrates that, although load balancing and striping are orthogonal strategies, they are not independent; rather well-tuned striping units and the cooling procedure exhibit synergetic effects. Note that, under the skewed load, the Gopt-λ method without cooling could not sustain a throughput of λ, as all our heuristic calculations for the derivation of striping units are based on uniform access frequencies (i.e., overly optimistic assumptions). Track striping, on the other hand, achieves a better load balance because of its finer striping units, but is still much inferior to the case with both tuned striping units and cooling.

In addition, to stress-test the responsiveness of “disk cooling” to dynamically evolving workloads, we generated a synthetic load, where the hot fraction of the data gradually moves across the entirety of files. This was implemented by dividing the total number of requests in the experiment into K phases, each with the same number of requests, and shifting the starting number of the heat-ranking list (i.e., the number of the hottest file) by N/K mod N (where N is the number of active files). Thus, while the workload characteristics are stable within each phase, a load shift occurs at each of the phase boundaries. In the experiments, one simulation run comprised 1,000,000 requests and K was set to 10; so each phase comprised 100,000 requests and the heat ranking was shifted by 200 files at the start of a new phase.

Table 7 shows the average response time results for this experiment as a function of the arrival rate λ for the Gopt140 striping unit (87 KB) without and with cooling. As in the previous experiment, the performance figures again indicate that the cooling method can effectively counteract the load imbalance, whereas the method without cooling suffers thrashing effects with response time approaching infinity for arrival rates higher than 120.

Figure 7 shows the response time and the cooling frequency as they vary over the duration of the experiment. In the two charts, the overall time period of the experiment is broken down into 100 equally sized intervals; each of the bars corresponds to an interval of length 83.3 s. The charts demonstrate that cooling is invoked particularly at the points when the hot files are shifted (namely, after every ten bars). At these points, the method without cooling suffers from particularly long disk queues, as some disks carry over their queue from the previous phase, while at the same time other disks start forming queues because of the shifted load imbalance. (These phenomena are, however, superimposed, to some extent, by the randomized nature of the synthetically generated load.) The cooling method, on the other hand, is fairly successful in eliminating the load imbalances and thus achieves response time improvements of more than a factor of 5 in this experiment.

In summary, application-specifically tuned striping in combination with cooling shows significant performance advantages over conventional methods. Not surprisingly, load imbalance is only an issue under high load, when queueing delays start becoming a factor. One may argue that an easy

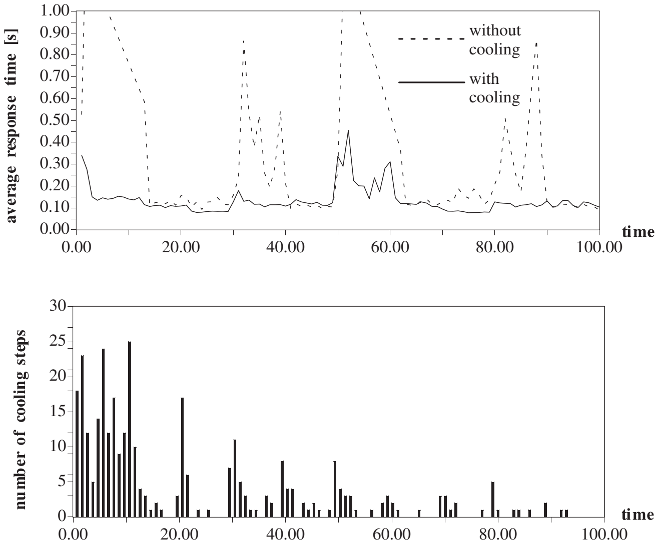

Fig. 7. Response time and cooling frequency for the dynamically evolving, skewed, synthetic workload varying over time

cure against load imbalance thherefore is to keep disk utilization low. However, for many applications, this implies unnecessarily high costs, as their performance requirements could be met with fewer disks at higher utilization. Furthermore, although system administration rules of thumb dictate that the disk utilization should generally be kept below 50%, this is often impossible during load peaks or when user demands grow faster than one can purchase additional disks. In fact, it is often exactly during load peaks, e.g., the Monday morning rush hour for retail banking or the hours right after an important sports event for a WWW server, when good response time matters most.

4.2 Experiments with application traces

To study the viability of the developed tuning procedures in a realistic application setting, we also conducted extensive experiments based on block access traces from a variety of applications, including online transaction processing, file systems, office document management, and WWW servers. Most of these experiments confirmed the results of the previous subsection. However, while such traces capture several essential characteristics of real-life application workloads (e.g., workload evolution over time, including transient load peaks), one has to be extremely careful about generalizing trace-based results. Traces constitute short-term snapshots with certain peculiarities that are not necessarily of fundamental nature. For this reason, we preferred deriving our basic performance results from a precisely controllable synthetic workload, as discussed in the previous subsection, and we restrict ourselves in this subsection to two sample results that were obtained with a WWW server trace and a trace from a bank's online transaction processing system.

4.2.1 World-Wide-Web server

This study is based on a trace that was recorded with the httpd logging facility on the WWW server ucmp1.berkeley.edu of the UC Museum of Paleontology at Berkeley

Table 8. Average response time in milliseconds for the WWW workload as a function of the request arrival rate λ

α	Track	Track-C
100	16.68	16.50
300	18.39	17.57
500	21.13	19.63
700	27.58	24.61
900	79.11	24.86
1000	203.36	28.57

over a time period of 120 h. Note that the fact that the requests were traced at the server site automatically factors out (client) caching. The trace contains 181,914 read accesses to an entirety of 9126 HTML and other files with heavily skewed access frequencies. The average request size was 14 KB, and the standard deviation of the requests size distribution was 28 KB.

We studied this trace under a spectrum of load levels. This was done by "speeding up" the arrivals in the original trace in the following way. Consider two requests r_i and r_{i+1} in the original trace which have an inter-arrival time of δ_i. Using a speed-up factor of α, the inter-arrival time between the requests becomes δ_i/α. Thus, in more general terms, if the original trace has an average inter-arrival time of $1/\lambda$, a trace with speed-up factor α has an average inter-arrival time of $1/(\lambda\alpha)$. Note that this method of "speeding up" a trace, albeit somewhat speculative, preserves all access characteristics of the original workload other than its arrival rate; particularly, the relative inter-arrival times between requests are preserved, which is essential to capture load bursts. The only case where the "speed-up" transformation would seriously distort the workload is when a large number of consecutive requests are correlated and must have a certain inter-arrival time. But this case is rather unlikely, given that a WWW server trace is typically based on a high number of concurrent users, each of which exhibits relatively long "think times", and the trace that we used showed this property.

Because of the small average request size and the moderate variance of request sizes, tuning the striping unit was not really an issue for this workload. Rather, the challenge in this trace was to cope well with the access skew in combination with the dynamic load fluctuations. So we concentrated on the impact of cooling, and compared a round-robin allocation for a striping unit of one track (i.e., 35 KB) without cooling, labeled "Track", versus the case with cooling, labeled "Track-C". Neither of the two cases exploited any a priori knowledge about the heat of files in the initial data allocation, hypothesizing that WWW server workloads exhibit dynamically evolving access patterns that cannot be statically predicted, so that manual tuning is ruled out. Cooling was invoked every 100 s of the original time scale (or, equivalently, every $100/\alpha$ s of the accelerated trace), with entire extents as migration units and an imbalance threshold of $\delta = 0.05$. Table 8 shows the average response time of Track versus Track-C as a function of the acceleration factor α.

Cooling exhibits noticeable performance even under medium load, and dramatically improves response time by an order of magnitude for the highest measured load. For $\alpha = 1000$, the average disk utilization was 24% and the uti-

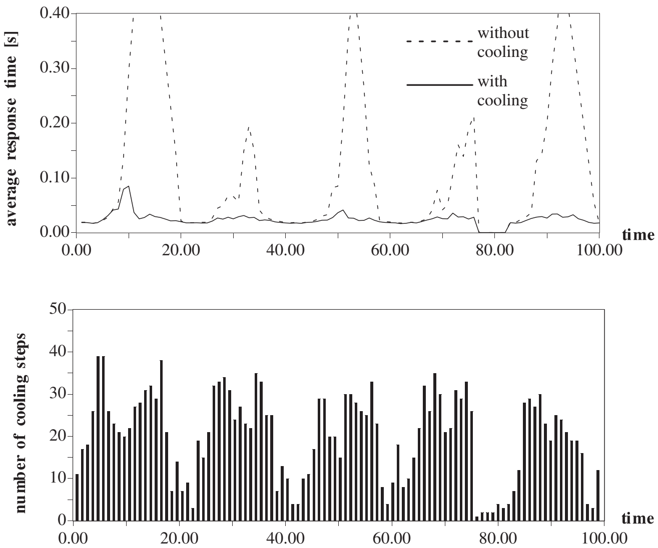

Fig. 8. Response time and cooling frequency for the WWW workload varying over time

lization of the hottest disk was 67% without cooling. With cooling, the average utilization increased slightly up to 25% because of the additional load incurred by data migrations, but the utilization of the hottest disk was reduced down to 39%, which accounted for the dramatic performance gain. Note that an average utilization of 25% appears to be a very light load; however, one has to take into account that the load fluctuates heavily over time, with very long disk queues built up during the load peaks. In terms of average disk queue lengths, the improvement by cooling was even more impressive: without cooling, the average queue length of the hottest disk (averaged over all points of time when a request was enqueued) was 63, whereas with cooling, this measure was 1.6 (i.e., 1 request in service and an expected value of 0.6 for the number of requests that wait in the queue). This effect is illustrated in Fig. 8, which shows the response time and the cooling frequency as they vary over the duration of the experiment, for the case of $\alpha = 1000$. In the two charts, the overall trace period is broken down into 100 equally sized intervals; so each of the bars corresponds to an interval of length 4.32 s in the accelerated trace. The improvement of response time due to cooling even exceeds a factor of 20 during the load peak.

4.2.2 Online transaction processing

A second study with real application workloads was based on an I/O trace from the OLTP system of a large Swiss bank (Union Bank of Switzerland). The database for this study consists of 166 files with a total size of 23 GB (but only a subset of these were accessed in the trace period). The I/O trace contains approximately 550,000 I/O requests to these files, recorded during 1 h. It was recorded at the disk controller level; so database caching is taken into account in this workload. As in a typical OLTP application, most requests read or write a single block (of size 8 KB in this application); the average request size is approximately 9 KB, with low variance. Thus, this workload does not warrant any specific tuning of the striping unit, so that we chose Track striping as the partitioning method. All files were allocated using a round-robin scheme, by first selecting a file's starting disk as the previous file's starting disk plus one modulo the number of disks, and then placing the file's striping units across the disks in a round-robin manner. The workload exhibits heavily skewed access frequencies both across files and within the hot files. In addition, the trace contains significant fluctuations in the access frequencies and in the overall arrival rate of requests.

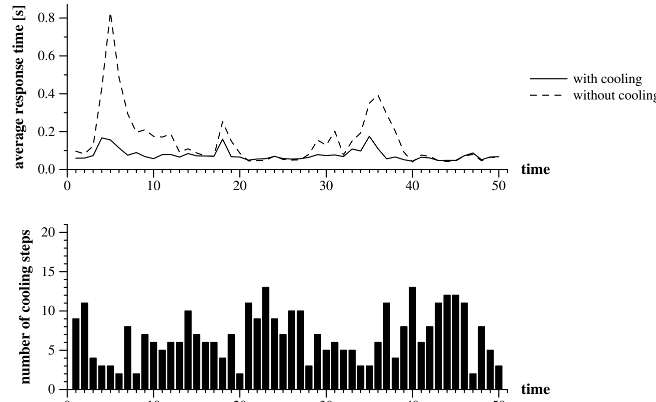

Fig. 9. Average response time and cooling frequency for the OLTP workload varying over time

We compared the performance of round-robin placement without cooling to round-robin allocation augmented with the cooling procedure. The cooling method improved the average response time of the requests by approximately a factor of 2 under high load.

As with the WWW experiment, we measured response time versus different "speed-up" factors of the arrival rate. The results in Fig. 9 are based on an arrival rate "speed-up" factor of 10. In the two charts, the overall trace period is broken down into 50 equally sized intervals; so each of the bars corresponds to an interval of length 7.2 s in the accelerated trace. As Fig. 9 shows, the cooling method could not improve response time in the initial light-load phase, since the load imbalance of the vanilla method did not yet incur any severe queueing. However, the cooling method did collect heat statistics during this phase. This enabled the cooling method to rebalance the disk load by data migration. Then, during the load peak (represented in Fig. 9 by the sharp increase of response time), the cooling method achieved a response time improvement by a factor of 5.3. Note that many OLTP applications have "soft" response time constraints such as ensuring a certain response time for 95% of the transactions. Thus, it is crucial to guarantee acceptable response time even during load peaks.

Figure 9 also shows the frequency of the data migration steps invoked by our cooling method, varying over time. The figure shows that our algorithm was careful enough so as not to initiate too many cooling steps during the high-load phases; rather, the data migrations were performed mostly during the low-load phases, thus improving the load balance for the next high-load phase at low cost. This robustness is achieved by explicitly trading off the benefit of cooling versus its additional cost, as discussed in Sect. 3.

5 Conclusion

5.1 Discussion of achievements and limitations

We have demonstrated the need for tuning the data placement in parallel disk systems, and we have presented various tuning heuristics for data partitioning, data allocation, and load balancing. The feasibility of the developed methods has been shown in a number of performance experiments, including simulations based on real-life traces.

We have developed an extended optimization procedure for file striping that explicitly takes into account throughput requirements and queueing delays, and in the process we have developed an analytical approximation to the well-known fork-join problem [62] in the specific setting of parallel disk systems. We have shown that our procedure for tuning the striping unit(s) of files is a very effective method for a wide spectrum of workloads, including multimedia information systems and other advanced database applications. Our optimization heuristics outperform all other striping methods for a specific target workload or higher, while being competitive also for loads lighter than the chosen target.

We believe that file-specific striping is important as a pre-requisite for incremental repartitioning of files, even if at best it provides marginal performance gains over global striping. Incremental repartitioning is crucial in order to cope with evolving performance requirements and to support system scalability. For example, when the throughput requirements of an application increase, we can repartition merely the hottest files in order to meet the new throughput goal, which is possible, since our approach supports file-specific striping units. Similarly, if more disks are added to a system, restriping of the most crucial files allows us to take advantage of the additional resources.

The methods for data allocation and redistribution complement the data-partitioning objective of minimizing queueing delays at the disks under heavy load by distributing the load across the disks as evenly as possible and by selectively redistributing the load dynamically by means of "disk-cooling" steps. Since our optimization procedure for data partitioning is based on uniform access frequencies, the combination of appropriately tuned striping and disk cooling is necessary to deal with skews in data access. By coupling these two procedures, our experiments have shown that at high loads we can obtain substantial performance gains. The dynamic load redistribution procedure has been shown to be efficient and robust, i.e., it performs disk cooling at a small cost and, very selectively, only during periods of low activity. We observe here that our procedures for data allocation and redistribution can be integrated with techniques for clustering the hottest files (extents) on each disk in its center [4, 69] and with disk-scheduling algorithms that reorder the requests in a queue (e.g., an "elevator" algorithm).

Some of the limitations of our approach are due to the restrictions we imposed on the workload characterization. As discussed earlier, our analytic approximation to the fork-join model is based on the assumption that the subrequests are uniformly distributed among the disks. Real workloads exhibit skewed access frequencies and both spatial and temporal correlations in the access patterns. An enhanced analytical model that would capture a broader class of workloads, e.g., one based on a Markov-chain traffic model, would clearly be very desirable. Similarly, our file-striping tuning method considers only the mean request sizes of the files and disregards the request size distribution. Incorporating all these additional workload aspects is very challenging in terms of their analytical tractability, but we believe that our model provides a good framework of reference for future work.

Our approach also assumes that the relevant workload parameters a priori can be estimated with sufficient accuracy. In the absence of appropriate input from application experts, the system itself must estimate these parameters by collecting online statistics. This implies that some parameter values can be determined only after the workload has been monitored for some time and only at that point can automatic tuning methods become effective. In such an environment, our data-partitioning method is limited to repartitioning of existing files, which we still believe is a very important issue.

Finally, we did not consider the impact of caching on disk load balancing. This simplification is justified for workloads whose distributions of access frequencies exhibit substantial skew even after eliminating the accesses to the files that reside in cache; this is the case, for example, for the two application-trace workloads (WWW server and OLTP system) that we used in our experiments. In other applications, however, it may well be the case that caching also can alleviate, if not completely eliminate, the imbalance in the disk load. A comprehensive treatment of such workloads requires an analytical understanding of how data placement on disks, dynamic data migration between disks, and the dynamic behavior of a cache interfere with each other, and is beyond the scope of this paper.

5.2 Future work

Our future work will focus on the following two major issues: combining the developed data placement methods with techniques for providing fault tolerance and high availability, and generalizing our approach towards shared-nothing parallel database systems and systems based on networks of workstations.

Our placement methods are orthogonal to the proposed fault tolerance techniques in that they can be combined, in a straightforward manner, with arbitrary variants of either mirroring (e.g., mirrored disks, interleaved declustering, or chained declustering [8, 15, 35, 64, 74]) or error-correcting codes (e.g., parity groups of some type [30, 31, 36, 37, 53, 54, 56–58, 60, 61, 66, 70]) or simply conventional logging [34]. However, the placement of data replicas or error-correcting information itself provides additional degrees of freedom that should be taken into account by an integrated approach in order to ensure the best possible performance and availability for given system costs [81].

In order to generalize our approach to a general shared-nothing parallel database system, we need to consider the impact of communication and CPU costs, in addition to the disk I/O service time. For the partitioning problem, the optimal partition size (e.g., the interval width in an interleaved range-partitioning scheme for relational data [28]) would again be derived from the optimal degree of parallelism, in analogy to our approach for striping. However, the operations under consideration are more complex (e.g., relational operators such as selection or join), and, in addition to latency and transfer time, the performance for a given degree of parallelism depends also on communication overhead and startup costs, as well as on the operations' CPU time consumption.

In all these considerations, an underlying assumption is that the system consists of homogeneous processing nodes. A further, even more challenging step would be to also consider heterogeneous systems, where the processing nodes can differ in their performance characteristics. i.e., processor speed, memory size, disk storage and performance capacity. Networks of workstations, also known as NOW [1], are evolving as a paradigm for high-performance computing. In order to make NOW a viable approach for large-scale data management, it is crucial to develop appropriate self-tuning and self-reliant data placement and storage techniques. A first approach along these lines, with specific consideration to load balancing, is presented in [77].

Acknowledgements. This research has been partially supported by NASA-Ames grant NAG2-846 grant, NSF grant IRI-9303583, and the ESPRIT LTR project 9141 (HERMES).

References

1. Anderson TE, Culler DE, Patterson DA, and the NOW Team (1995) A Case for NOW (Networks of Workstations). IEEE Micro 15: 54–64
2. Awerbuch B, Bartal Y, Fiat A (1993) Competitive Distributed File Allocation. ACM Symposium on Theory of Computing, pp 164–173
3. Azar Y, Naor J, Rom R (1992) The Competitiveness of Online Assignment. 3rd ACM/SIAM Symposium on Discrete Algorithms
4. Akyürek S, Salem K (1995) Adaptive Block Rearrangement, ACM Trans Comput Sys 13(2): 89–121
5. Bartal Y, Fiat A, Karloff H, Vohra R (1992) New Algorithms for an Ancient Scheduling Problem. Proceedings of the 24th ACM Symposium on Theory of Computing, pp 51–58
6. Bartal Y, Fiat A, Rabani Y (1992) Competitive Algorithms for Distributed Data Management. ACM Symposium on Theory of Computing
7. Bestavros A (1995) Demand-based Document Dissemination to Reduce Traffic and Balance Load in Distributed Information Systems. Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing
8. Bitton D, Gray JN (1988) Disk Shadowing. Proceedings of the 14th International Conference on Very Large Data Bases, pp 331–338
9. Chamberlin DD, Schmuck FB (1992) Dynamic Data Distribution (D^3) in a Shared-Nothing Multiprocessor Data Store. International Conference on Very Large Data Bases, Vancouver, pp 163–174
10. Chen PM, Lee EK, Gibson GA, Katz RH, Patterson DA (1994) RAID: High-Performance, Reliable Secondary Storage. ACM Comput Surv 26(2): 145–185
11. Chen PM, Lee EK (1995) Striping in a RAID Level 5 Disk Array. ACM SIGMETRICS Conference, pp 136–145
12. Chen PM, Patterson DA (1990) Maximizing Performance in a Striped Disk Array. Proceedings of the 17th International Symposium on Computer Architecture(SIGARCH), pp 322–331
13. Chen S, Towsley D (1993) The Design and Evaluation of RAID 5 and Parity Striping Disk Array Architectures, J Parallel Distrib Comput 17(1): 58–74
14. Copeland G, Alexander W, Boughter E, Keller T (1988) Data Placement in Bubba, Proceedings of the SIGMOD International Conference on Management of Data, pp 99–108
15. Copeland G, Keller T (1989) A Comparison of High-Availability Media Recovery Techniques. Proceedings of the SIGMOD International Conference on Management of Data, pp 98–109
16. Copeland G, Keller T, Smith M (1992) Database Buffer and Disk Configuring and the Battle of the Bottlenecks. International Workshop on High-Performance Transaction Systems
17. Coffman Jr. EG, Garey MR, Johnson DS (1978) An Application of Bin-Packing to Multiprocessor Scheduling. SIAM J Comput 7(1): 1–17
18. DeWitt DJ, Gray JN (1992) Parallel Database Systems: The Future of High Performance Database Systems. Commun ACM 35(6): 85–98
19. Dowdy W, Foster DV (1982) Comparative Models of the File Assignment Problem. ACM Comput Surv 14(2): 287–313
20. Du HC, Sobolewski JS (1982) Disk Allocation for Cartesian Product Files on Multiple Disk Systems. ACM Trans Database Syst 7(1): 82–101
21. Faloutsos C, Metaxas D (1991) Disk Allocation Methods Using Error Correcting Codes. IEEE Trans Comput 40(8): 907–914
22. Faigle U, Kern W, Turan G (1989) On the Performance of On-line Algorithms for Particular Problems. Acta Cybernetica 9: 107–119
23. Fatto L, Hahn S (1984) Two Parallel Queues Created By Arrivals With Two Demands I. SIAM J Appl Math 44: 1041–1053
24. Fatto L (1985) Two Parallel Queues Created By Arrivals With Two Demands II. SIAM J Appl Math 45: 861–878
25. Ganger GR, Worthington BL, Hou RY, Patt YN (1994) Disk Arrays: High-Performance, High-Reliability Storage Subsystems. IEEE Comput 27(3): 30–36
26. Garey MR, Johnson DS (1979) Computers and Intractability. W.H. Freeman
27. Ghandeharizadeh S, DeWitt DJ (1990) A Multiuser Performance Analysis of Alternative Declustering Strategies. 6th IEEE International Conference on Data Engineering, Los Angeles, pp 466–475
28. Ghandeharizadeh S, DeWitt DJ (1990) Hybrid-range Partitioning Strategy: A New Declustering Strategy for Multiprocessor Database Machines. 16th International Conference on Very Large Data Bases, Brisbane, pp 481–492
29. Ghandeharizadeh S, DeWitt DJ (1994) MAGIC: A Multiattribute Declustering Mechanism for Multiprocessor Database Machines. IEEE Trans Parallel Distrib Syst 5(5): 509–524
30. Gibson GA, Hellerstein L, Karp RM, Katz RH, Patterson DA (1989) Failure Correction Techniques for Large Disk Arrays. Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems, pp 123–132
31. Gibson GA (1992) Redundant Disk Arrays: Reliable, Parallel Secondary Storage. MIT Press, Cambridge, Mass.
32. Graham RL (1969) Bounds on Certain Multiprocessing Anomalies. SIAM J Appl Math 17: 416–429
33. Gray JN, Horst B, Walker M (1990) Parity Striping of Disk Arrays: Low-Cost Reliable Storage with Acceptable Throughput. Proceedings of the 16th International Conference on Very Large Data Bases, pp 148–161
34. Gray J, Reuter A (1993) Transaction Processing: Concepts and Techniques. Morgan Kaufmann, San Mateo, Calif.
35. Hsiao H, DeWitt DJ (1993) A Performance Study of Three High-Availability Data Replication Strategies. Int J Distrib Parallel Databases 1(1): 53–79
36. Holland M, Gibson GA (1992) Parity Declustering for Continuous Operation in Redundant Disk Arrays. Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems, pp 23–35
37. Holland M, Gibson GA, Siewiorek DP (1994) Architectures and Algorithms for On-Line Failure Recovery in Redundant Disk Arrays. Distrib Parallel Databases 2(3): 295–335
38. Hua KA, Lee C, Young HC (1993) Data Partitioning for Multicomputer Database Systems: A Cell-based Approach. Inf Syst 18(5): 329–342
39. Jain R (19921) The Art of Computer Systems Performance Analysis. Wiley, New York
40. Karger DR, Phillips SJ, Torng E (1994) A Better Algorithm for an Ancient Scheduling Problem. 5th ACM/SIAM Symposium on Discrete Algorithms

41. Katz RH, Gibson GA, Patterson DA (1989) Disk System Architectures for High Performance Computing. Proc IEEE 77(12): 1842–1858
42. Katz RH, Hong W (1993) The Performance of Disk Arrays in Shared Memory Database Machines. Distrib Parallel Databases 1(2): 167–198
43. Kim MY (1986) Synchronized Disk Interleaving. IEEE Trans Comput C-35(11): 978–988
44. Kim MY, Pramanik S (1988) Optimal File Distribution for Partial Match Retrieval. Proceedings of the SIGMOD International Conference on Management of Data, pp 173–182
45. Kim MY, Tantawi AN (1991) Asynchronous Disk Interleaving: Approximating Access Delays. IEEE Trans Comput 40(7): 801–810
46. Kleinrock L (1975) Queueing Systems. Wiley, New York
47. Knuth DE (1973) The Art of Computer Programming. Vol. 3: Sorting and Searching. Addison-Wesley, Reading, Mass.
48. Lee EK, Katz RH (1993) An Analytic Performance Model of Disk Arrays. Proceedings of the International Conference on Measurement and Modeling of Computer Syst (ACM SIGMETRICS), pp 98–109
49. Lee L-W (1994) Optimization of Load-Balanced File Allocation. Doctoral Thesis, Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, Ill.
50. Litwin W, Neimat M-A, Schneider DA (1993) LH* – Linear Hashing for Distributed Files. ACM SIGMOD Conference, Washington, pp 327–335
51. Livny M, Khoshafian S, Boral H (1987) Multi-Disk Management Algorithms. Proceedings of the International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS), pp 69–77
52. Lui JCS, Muntz RR, Towsley D (1994) Computing Performance Bounds for Fork-Join Queueing Models. Technical Report 940034, Computer Science Department, UCLA
53. Menon J (1994) Performance of RAID5 Disk Arrays with Read and Write Caching. Distrib Parallel Databases 2(3): 261–293
54. Menon J, Cortney J (1993) The Architecture of a Fault-Tolerant Cached RAID Controller. Proceedings of the 20th Symposium on Computer Architecture (ACM SIGARCH), pp 76–86
55. Mohan C, Pirahesh H, Tang WG, Wang Y (1994) Parallelism in Relational Database Management Systems. IBM Syst J 33(2): 349–371
56. Menon J, Roche J, Kasson J (1993) Floating Parity and Data Disk Arrays. J Parallel Distrib Comput 17(1): 129–139
57. Muntz RR, Lui JCS (1990) Performance Analysis of Disk Arrays Under Failure. Proceedings of the 16th International Conference on Very Large Data Bases, pp 162–173
58. Merchant A, Yu PS (1992) Design and Modeling of Clustered RAID. Proceedings of the 22nd Annual Symposium on Fault-Tolerant Computing, pp 140–149
59. Merchant A, Yu PS (1995) Analytic Modeling and Comparisons of Striping Strategies for Replicated Disk Arrays. IEEE Trans Comput 44(3): 419–433
60. Mogi K, Kitsuregawa M (1994) Dynamic Parity Stripe Reorganizations for RAID5 Disk Arrays. Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems, Austin, pp 17–26
61. Mogi K, Kitsuregawa M (1995) Hot Block Clustering for Disk Arrays with Dynamic Striping. Proceedings of the 21st International Conference on Very Large Data Bases, Zurich, pp 90–99
62. Nelson R, Tantawi AN (1988) Approximate Analysis of Fork/Join Synchronization in Parallel Queues, IEEE Transactions on Comput 37(6): 739–743
63. Patt YN (Guest Editor) (1994) The I/O Subsystem: A Candidate for Improvement. IEEE Comput 27(3)
64. Polyzois CA, Bhide A, Dias DM (1993) Disk Mirroring with Alternating Deferred Updates. Proceedings of the 19th International Conference on Very Large Data Bases, pp 604–617
65. Patterson DA, Gibson GA, Katz RH (1988) A Case for Redundant Arrays of Inexpensive Disks (RAID). Proceedings of the SIGMOD International Conference on Management of Data, pp 109–116
66. Reddy ALN, Chandy J, Banerjee P (1993) Design and Evaluation of Gracefully Degradable Disk Arrays. J Parallel Distrib Comput 17(1): 28–40
67. Ruemmler C, Wilkes J (1994) An Introduction to Disk Drive Modeling. IEEE Comput 27(3): 17–28
68. Salem K, Garcia-Molina H (1986) Disk Striping. Proceedings of the 2nd International Conference on Data Engineering, pp 336–342
69. Staelin C, Garcia-Molina H (1990) Clustering Active Disk Data to Improve Disk Performance. Technical Report CS-TR-283-90, Department of Computer Science, Princeton University, N.J.
70. Stodolsky D, Gibson G, Holland M (1993) Parity Logging: Overcoming the Small Write Problem in Redundant Disk Arrays. Proceedings of the 20th Symposium on Computer Architecture (ACM SIGARCH), pp 64–75
71. Scheuermann P, Weikum G, Zabback P (1992) Automatic Tuning of Data Placement and Load Balancing in Disk Arrays. In: Database Syst for Next-Generation Applications – Principles and Practice. Advanced Database Research and Development Series, World Scientific Publications, pp 291–301
72. Scheuermann P, Weikum G, Zabback P (1993) Adaptive Load Balancing in Disk Arrays. Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms (FODO), pp 345–360
73. Schwetman H (1992) CSIM Reference Manual (Revision 16). MCC Technical Report ACT-ST-252-87, Microelectronics and Computer Technology Corporation, Austin, Tx.
74. Solworth JA, Orji CU (1993) Distorted Mapping Techniques to Achieve High Performance in Mirrored Disk Systems. Int J Distrib Parallel Databases 1(1): 81–102
75. Stonebraker M, Aoki PM, Pfeffer A, Sah A, Sidell J, Staelin C, Yu A (1996) Mariposa: a wide-area distributed database system, VLDB J 5(1): 48–63
76. Vingralek R, Breitbart Y, Weikum G (1994) Distributed File Organization with Scalable Cost/Performance. ACM SIGMOD Conference, Minneapolis, pp 253–264
77. Vingralek R, Breitbart Y, Weikum G (1995) SNOWBALL: Scalable Storage on Networks of Workstations with Balanced Load. Technical Report, Department of Computer Science, University of Kentucky
78. Weikum G, Hasse C, Mönkeberg A, Zabback P (1994) The COMFORT Automatic Tuning Project. Inf Syst 19(5): 381–432
79. Weikum G, Zabback P, Scheuermann P (1990) Dynamic File Allocation in Disk Arrays. Proceedings of the SIGMOD International Conference on Management of Data, pp 406–415; extended version available as: Technical Report No. 147, Computer Science Dept., ETH Zürich
80. Weikum G, Zabback P (1992) Tuning of Striping Units in Disk-Array-Based File Systems. Proceedings of the 2nd International Workshop on Research Issues on Data Engineering: Transaction and Query Processing (RIDE-TQP), pp 80–87
81. Wilkes J, Golding R, Staelin C, Sullivan T (1995) The HP AutoRAID Hierarchical Storage System. Proceedings of the 15th ACM Symposium on Operating Systems Principles
82. Wolf J (1989) The Placement Optimization Program: A Practical Solution to the Disk File Assignment Problem. Proceedings of the International Conference on Measurement and Modeling of Computer Syst (ACM SIGMETRICS), pp 1–10
83. Wolfson O, Jajodia S (1992) Distributed Algorithms for Dynamic Replication of Data. Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Syst (PODS), pp 149–163
84. Zabback P (1994) I/O Parallelism in Database Systems – Design, Implementation, and Evaluation of a Storage System for Parallel Disks. (in German) Doctoral Thesis, Department of Computer Science ETH Zürich, ETH-NR.: 10629