First Workshop on Algorithms and Architectures for Self-Managing Systems.
In conjunction with Federated Computing Research Conference (FCRC). San Diego CA. June 11, 2003.

A Two-Tiered Software Architecture for Automated Tuning of Disk Layouts

Brandon Salmon, Eno Thereska, Craig A.N. Soules, Gregory R. Ganger
Carnegie Mellon University

ABSTRACT

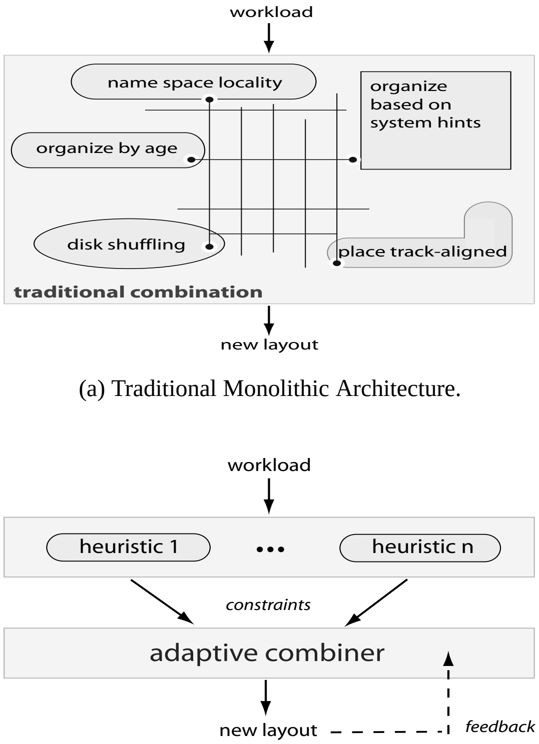
Many heuristics have been developed for adapting on-disk data layouts to expected and observed workload characteristics. This paper describes a two-tiered software architecture for cleanly and extensibly combining such heuristics. In this architecture, each heuristic is implemented independently and an adaptive combiner merges their suggestions based on how well they work in the given environment. The result is a simpler and more robust system for automated tuning of disk layouts, and a useful blueprint for other complex tuning problems such as cache management, scheduling, data migration, and so forth.

1. INTRODUCTION

Internal system policies, such as on-disk layout and disk prefetching, have been the subject of decades of research. Researchers try to identify algorithms that work well for different workload mixes, developers try to decide which to use and how to configure them, and administrators must decide on values for tunable parameters (e.g., run lengths and prefetch horizons). Unfortunately, this process places significant burden on developers and administrators, yet still may not perform well in the face of new and changing workloads.

To address this, researchers now strive for automated algorithms that learn the right settings for a given deployed system. Of course, different policy+parameter configurations work best for different workloads meaning that any particular setup will work well for one workload and poorly for others. Worse, most deployed systems support many workloads simultaneously, potentially making any single decision suboptimal for the aggregate. Devising a composite algorithm for such circumstances can be a daunting task, and updating such an algorithm even more so.

This paper describes a two-tiered architecture for such automated self-tuning software, using on-disk data layout as a concrete example. Instead of a single monolithic algorithm, as illustrated in Figure 1a, the decision-making software consists of a set of independent *heuristics* and an *adaptive combiner*, used for merging the heuristics' suggested solutions as shown in Figure 1b. Each heuristic implements a single policy that hopefully works well in some circumstances but not necessarily in all. Heuristics provide suggested *constraints* on the end layout, such as placing a given block in a given region or allocating a set of blocks sequentially. The adaptive combiner uses prediction models and on-line observations to balance and merge conflicting constraints.



(a) Traditional Monolithic Architecture.

(b) Two-tiered Architecture.

Figure 1: Two-tiered vs. traditional architecture for adaptive layout software. The traditional architecture combines different heuristics in an ad-hoc fashion, usually using a complicated mesh of if-then-else logic. The two-tiered architecture separates the heuristics from the combiner and uses feedback to refine its decisions and utilize the best parts of each heuristic.

This two-tiered architecture provides three benefits. First, heuristic implementations can focus on particular workload characteristics, making local decisions without concern for global consequences. Second, new heuristics can be added easily, without changing the rest of the software; in fact, bad (or misimplemented) heuristics can even be handled, because their constraints will be identified as less desirable and ignored. Third, the adaptive combiner can balance constraints without knowledge of or concern for how they are generated. The overall result is a simpler and more robust software structure.

This paper describes an instance of this two-tiered architecture and its role in an automated system for on-disk layout reorganization. Promising initial results are presented and questions being explored in ongoing work are discussed.

2. RELATED WORK

The AI community continues to develop and extend the capabilities of automated learning systems. The systems community is adopting these automated approaches to address hard problems in systems management. This section briefly discusses relevant related work, both from the AI and systems perspectives.

Related AI Work: The AI community has long recognized the need for self-managing systems. In fact, a whole branch of AI research, machine learning, exists especially to solve real-life problems where human involvement is not practical [14].

One general AI problem of relevance is the *n-experts problem*, in which a system must choose between the outputs of n different experts. The n-experts problem is not an exact match to our problem, because we are merging experts' suggestions rather than picking one. Nonetheless, solutions such as the weighted majority algorithm [9] provide valuable guidance.

Another general challenge for the AI community is the exploration of a *state space* (i.e., the set of all possible solutions to an optimization problem). For example, our current prototype explores its state space using a guided hill-climbing algorithm and a method similar to simulated annealing to avoid local maxima.

Adaptive Disk Layout Techniques: A disk layout is the mapping between a system's logical view of storage and physical disk locations. Useful heuristics have been devised based on block-level access patterns and file-level information.

Block-based heuristics arrange the layout of commonly accessed blocks to minimize access latency. For example, Ruemmler and Wilkes [19] explored putting frequently used data in the middle of the disk to minimize seek time. Wang and Hu [25] tuned a log-structured file system [17] by putting active segments on the high bandwidth areas of the disk. Several researchers [1, 10, 15, 26] have explored replication of disk blocks to minimize seek and rotational latencies.

File-based heuristics use information about file inter- and intra-relationships to co-locate related blocks. For example, most file systems try to allocate blocks of a file sequentially. C-FFS allocates adjacently the data blocks that belong to multiple small files named by the same directory [5]. Hummingbird and Cheetah perform similar grouping for related web objects (e.g., an HTML document and its embedded images) [8, 23].

Other system management policies: Relevant research has also been done on storage system policies, such as caching and prefetching [2, 7, 16]. Most notably, Madhyastha and Reed [12] explore a system for choosing one of several file caching policies based on access patterns. Unlike our system, however, their system works to determine which single heuristic to use for the particular workload, rather than combining the suggestions of multiple heuristics. Similar schemes have been used in other domains as well, such as branch prediction in modern processors [13, 24].

3. TWO-TIERED LEARNING FOR LAYOUT

This work focuses on the problem of identifying a disk layout that improves performance for a given workload. At the most general level, this is a learning problem that takes as input a workload and outputs a new layout. However, due to the size of the state space, solving this problem using a monolithic learning algorithm is intractable; a typical disk has millions of blocks, and workloads often contain millions of requests.

The two-tiered learning architecture addresses this problem by using heuristics to build up a smaller, more directed state space. The system then searches for better performing disk layouts within this smaller space.

3.1 Overview

Figure 2 illustrates the two-tiered learning architecture. The constraint generation layer consists of a collection of independent heuristics. Each of these heuristics generates a set of constraints based on the workload. The adaptive combiner consists of three parts: the learner, the layout manager, and the performance analyzer.

The *learner* assigns weights to each of the constraints based on the performance of previous disk layouts. It then uses the weights to resolve any conflicts between constraints, creating a single set of consistent constraints.

The *layout manager* takes the constraint set from the learner and builds a new disk layout.

The *performance analyzer* takes each candidate layout and determines its success based on the target performance metrics and the given workload. The results of the performance analyzer are then passed to the learner for use in updating the constraint weights.

In order to search the state space for the maximum allowed time, the adaptive combiner holds the best observed layout. If at any time the adaptive combiner must be stopped (e.g., due to computation constraints or diminishing returns), it can immediately output the best observed layout.

The remainder of this section discusses the constraint language, how example heuristics map to this language, conflict resolution between weighted constraints, and how weights are generated.

3.2 Common Constraint Language

The learner uses constraints as the common language of disk organization heuristics. A constraint is an invariant on the disk layout that a heuristic considers important. A constraint may be specified

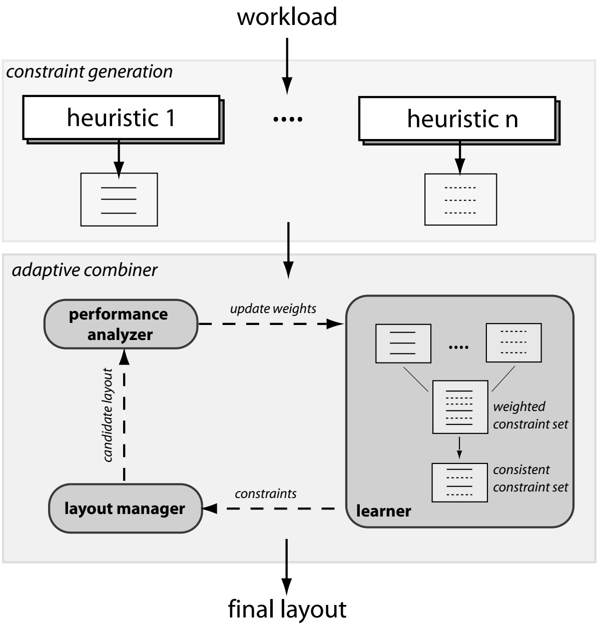


Figure 2: The two-tiered learning architecture. This figure illustrates the two components of a two-tiered architecture for refining disk layout. The *constraint generation* layer consists of the individual heuristics and their constraints. The *adaptive combiner* layer merges the constraints and refines the resulting data layout.

on a single disk block or a set of blocks. The learner combines heuristics by selectively applying their constraints.

Table 1 shows three example constraints. *Place-in-region* constraints specify in which region of the disk a set of blocks should be placed.[1] *Place-seq* constraints specify a set of blocks that should be placed sequentially. *Place-as-set* constraints specify a set of blocks that should be placed adjacent and fetched as a single unit. These three constraint types have been sufficient for many heuristics, but additions are expected for some future heuristics. For example, heuristics that exploit replication and low-level device characteristics (e.g. track-aligned extents [22]) may require additions.

3.3 Heuristics

For illustration, this section describes five heuristics currently used in the prototype constraint generation layer.

Disk Shuffling: The disk shuffling heuristic generates *place-in-region* constraints to place frequently accessed data near the middle of the disk and less frequently accessed data towards the edges [19]. Such an arrangement reduces the average seek distance of requests.

Sequentiality: The sequentiality heuristic generates *place-seq* constraints for sets of blocks usually accessed in a sequential manner. Doing so exploits the efficiency of disk streaming.

Streaming: The streaming heuristic generates *place-in-region* constraints to place blocks fetched in large sequential requests on the outside tracks of the disk. This utilizes the higher streaming bandwidth of the outer disk tracks.

Constraint Type	Description
place-in-region	place blocks into a specific region
place-seq	place blocks sequentially
place-as-set	place blocks adjacent and fetch as a unit

Table 1: Common Constraint Language. This table shows three example constraints we use in our implementation.

Bad: The “Bad” heuristic generates *place-in-region* constraints to spread the most frequently accessed blocks across the disk in an attempt to destroy locality. It exists to test the learner’s ability to avoid poorly performing constraints.

Default: This heuristic places all blocks in their original locations.

3.4 Conflict Resolution

Because of their independence, different heuristics may generate conflicting constraints (e.g., the streaming heuristic places blocks on the outer tracks, while disk shuffling places them near the center of the disk). The learner must resolve these conflicts, choosing which constraints to apply and which to ignore.

Intuitively, some constraints will be more effective than others. To represent this, the learner assigns a weight to each constraint, where a higher weight implies greater effectiveness. Section 3.5 discusses weight generation.

To maximize effectiveness of the final layout, conflict resolution tries to maximize the sum of applied constraints’ weights. The learner uses a three-step algorithm to do so. First, the learner sorts the constraints in descending order by weight. Second it randomly reorders some of the constraints. Third, it greedily applies as many constraints as possible, starting at the highest weighted constraint.

The randomization in the second step is a standard technique used to keep the learner from getting stuck in local minima [20]. The learner starts with a high amount of randomization and decreases the randomization as it narrows in on a solution.

Although the above approach has worked reasonably during initial testing, in general this is a constraint satisfaction problem. Examining the variety of algorithms that provide more accurate solutions [4] is an area of ongoing work.

3.5 Learning Weights

The adaptive combiner uses average block response time as its performance metric. Thus, weights should increase as response time decreases and increase as the number of requests increases. The learner uses the following function to compute the weight of each constraint:

[1] The system currently divides the disk into 24 regions to simplify placement constraints.

$$w_c = \frac{\sum_{b \in B_c} \sum_{a \in A_b} (resp_{avg} - resp_a)}{|B_c|} \quad (1)$$

w_c = computed weight of constraint c
B_c = set of blocks in constraint c
A_b = set of accesses to block b
b = a block in constraint c
a = an access to block b
$resp_{avg}$ = average response time of the best performing layout
$resp_a$ = response time for access a

For each block in constraint c, Equation (1) sums the response time improvement for each access to the block, thus favoring lower response times and more accesses. It then normalizes this value across all the blocks in the constraint so that larger constraints are not favored.

Unfortunately, the weight generated by Equation (1) is not independent of the other constraints applied in the layout. For example, the access time for request n will depend in part on the location accessed by request $n-1$, which may have been placed by a distinct constraint. Because of such dependencies, the adaptive combiner iterates on the disk layout, refining the weights toward a more accurate value. At each iteration, it generates a new layout, evaluates it, and updates the weight of each constraint using the following equation:

$$w_{c,i+1} = (1-\alpha)w_{c,i} + \alpha w_c \quad (2)$$

On each iteration, the learner first calculates w_c using Equation (1). It then combines that result with the weight of the previous iteration using Equation (2). Over a large number of iterations, the weights of poorly performing constraints will decrease, but a single instance of poor performance will not permanently reject a constraint. Increasing α may decrease the possibility of converging, while decreasing α raises the chance of falling into a local minima.

4. CONTINUOUS REORGANIZATION

The two-tiered learning architecture described in Section 3 is one part of a system for continuously tuning disk layout. Figure 3 shows the additional infrastructure required to feed the learning architecture and make use of its output. This section discusses the four components of our prototype for continuous reorganization of disk layouts.

Tracer: On the critical path, the *tracer* records I/O requests as they are sent to the disk. Both the heuristics and the performance analyzer use the traced stream of requests as input. In our current implementation, the heuristics use only block-based requests. Future heuristics may utilize file system information as well.

Mapper: Also on the critical path is the *mapper*. The mapper translates logical disk locations to physical locations, allowing the disk layout to be modified transparently to the host.

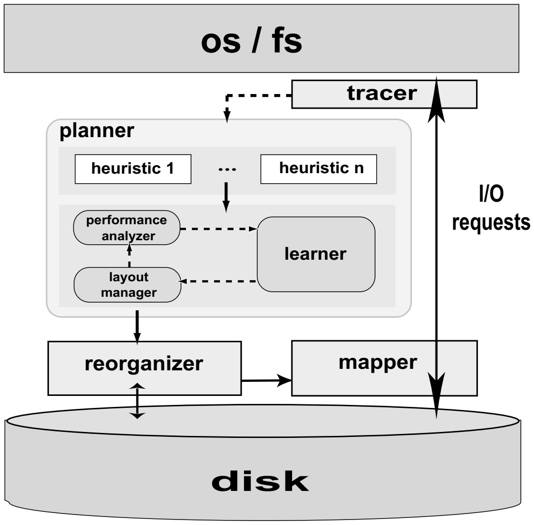


Figure 3: Continous Reorganization. This figure illustrates our prototype, that uses the two-tiered learning architecture to continously reorganize the on-disk layout.

Planner: The *planner* is our implementation of the learning architecture described in Section 3. The planner implements heuristics as individual C++ objects, allowing it to add or subtract heuristics easily. To avoid the unsolved problem of workload characterization, the performance analyzer feeds I/O traces into DiskSim [3] to model disk layout performance. The advantage of trace-driven simulation over analytical models based on workload characterization is that simulation has been shown to be able to capture all aspects of a request stream.

Reorganizer: The *reorganizer* reorganizes the current disk layout to match the new layout produced by the planner. One goal of the reorganizer is to minimize the impact on the foreground workload during the layout rearrangement. Towards that goal, the reorganizer module exploits idle disk time [6] and freeblock scheduling [11] to do its block rearrangements.

Because these four components are logically disconnected from both the OS and the disk, they can be implemented wherever is most appropriate (e.g., in the file system or in disk firmware.)

5. EVALUATION

This section presents early results from our prototype to illustrate some of the architecture's features. We compare four configurations to the original, unmodified layout of disk blocks[2]: disk shuffling alone (*shuffling*), the sequentiality heuristic in combination with the streaming heuristic (*sequential*), the system with the previous three good heuristics (*"cr w/out bad"*), and the system with the three good heuristics and the Bad heuristic (*"cr w/ bad"*.) We also evaluated the Bad heuristic on its own, but we do not show the results because the average response times were consistently orders of magnitude longer than the original trace.

We present results for these configurations on the first week of

[2] Each "block" is a disk-level unit 512 bytes in size, identified by the logical block number (LBN) assigned by the OS.

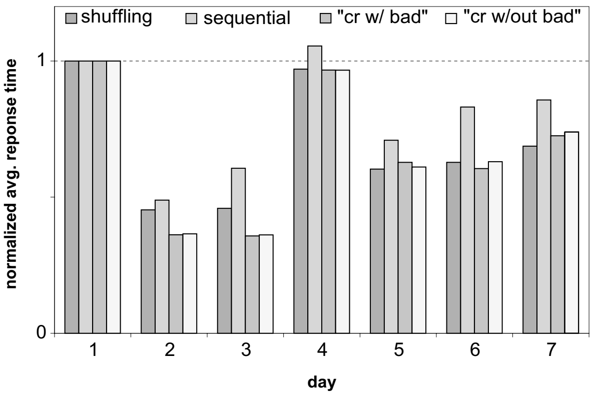


Figure 4: Response time comparison. This figure shows the average response time of our four configurations normalized to the unmodified disk layout.

disk 5 of the cello92 trace [18]. For each day of the trace, we generated a disk layout by running 50 iterations of the planner, and evaluated the new layout on the next day's trace. We repeated the process for each of the 7 days. To simulate the disk, we used the DiskSim simulator with parameters calibrated for a 9GB Quantum Atlas10K disk drive extracted using the DIXTrac tool [21].

Figure 4 shows the reduction in response time from the base case for the four configurations on each of the 7 days. For each day, the average response time is normalized to the corresponding day's performance when using the unmodified disk layout. Four points are worthy of note. First, throughout the trace, continuous reorganization stays close to the best performing heuristic. Second, no improvement is seen on day 1, because this is the "training" day before the first reorganization. Third, merging of heuristics provides better results than any single heuristic for days 2 and 3, illustrating one promise of the two-tiered architecture. Fourth, the adaptive combiner successfully avoids being hurt by the Bad heuristic. In fact, including the Bad heuristic often provides a slight benefit — while the Bad heuristic performs poorly in general, a few of its constraints turn out to be useful (quite unexpectedly), and the system finds and implements them.

Although preliminary, these results indicate that continuous reorganization has the potential to merge the best characteristics of a variety of heuristics while avoiding penalties associated with bad heuristics.

6. ONGOING WORK

As this work moves forward, we continue to add more heuristics and to refine the different components of the prototype. This section identifies some challenges facing our two-tiered software model and discusses possible solutions.

Exploring state space: The heuristics used, although independent from the point of view of the constraint generator layer, are interdependent from the point of view of the adaptive combiner. To handle these dependencies, the learner explores the state space by continuously refining the set of weights on the constraints the heuristics generate. It is not yet clear how big a role the dependencies among heuristics play. In general, we believe that the more dependent the heuristics are on each other, the more iterations are needed to determine the right constraint weights.

We could also train a neural network to generate constraint weights from block statistics gathered from the trace instead of using equations (1) and (2). This approach would allow us to guess the weight of a constraint without actually evaluating a layout containing that constraint. However, it may require a large number of statistics and many iterations to train a network. Initial experiments with neural networks showed that, even after hundreds of iterations, the networks were not converging on good approximations of the values.

Conflict resolution: A more sophisticated conflict resolution algorithm may provide a better combination of constraints, and could eliminate the need to normalize weights across constraint length. Further work will explore different approaches to solving the problem, and their effect on performance.

Minimizing block movement: The learning architecture presented in Section 3 does not consider the cost of moving a block as it attempts to improve the disk layout. A complete system should take block movement cost into account. Idle time and freeblock scheduling [11] can be used to do reorganization without impact on foreground workloads, but at a cost in reorganization time.

Update patterns: Standard file systems overwrite data in place, while snapshots and other non-overwrite mechanisms require that new data go to unallocated locations. A complete system should account for the different update patterns associated with different workloads.

7. SUMMARY

The two-tiered software architecture cleanly and adaptively combines many disk layout heuristics, achieving the best properties of each and avoiding the worst. We believe that this same architecture can be applied successfully to other long-standing policy decisions, such as cache management and inter-device data placement.

Acknowledgements

We thank Shobha Venkataraman and James Hendricks for their help in developing this system. We thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support.

8. REFERENCES

[1] S. Akyurek and K. Salem. Adaptive block rearrangement. *ACM Transactions on Computer Systems*, **13**(2):89–121. ACM Press, May 1995.

[2] I. Ari, A. Amer, R. Gramacy, E. L. Miller, S. A. Brandt, and D. D. E. Long. ACME: Adaptive Caching Using Multiple Experts. Workshop on Distributed Data and Structures, 2002.

[3] J. S. Bucy and G. R. Ganger. *The DiskSim simulation environment version 3.0 reference manual*. Technical Report CMU–CS–03–102. Department of Computer Science Carnegie-Mellon University, Pittsburgh, PA, January 2003.

[4] E. C. Freuder and R. J. Wallace. Partial constraint satisfaction. *Artifical Intelligence*, **58**(1-3):21–70. Elsevier Science, 1992.

[5] G. R. Ganger and M. F. Kaashoek. Embedded inodes and explicit grouping: exploiting disk bandwidth for small files.

USENIX Annual Technical Conference, pages 1–17, January 1997.

[6] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes. Idleness is not sloth. Winter USENIX Technical Conference, pages 201–212. USENIX Association, 1995.

[7] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. Summer USENIX Technical Conference, pages 197–207. USENIX Association, 1994.

[8] M. F. Kaashoek, D. R. Engler, G. R. Ganger, and D. A. Wallach. Server operating systems. ACM SIGOPS. European workshop: Systems support for worldwide applications, pages 141–148. ACM, 1996.

[9] N. Littlestone and M. K. Warmuth. *The weighted majority algorithm*. UCSC–CRL–89–16. DEPTCS,. University of California at Santa Cruz, July 1989.

[10] S.-L. Lo. *Ivy: A study on replicating data for performance improvement*. HPL–CSP–90–48. Concurrent Systems Project, Hewlett-Packard Laboratories, 14 December 1990.

[11] C. R. Lumb, J. Schindler, G. R. Ganger, D. F. Nagle, and E. Riedel. Towards higher disk head utilization: extracting free bandwidth from busy disk drives. Symposium on Operating Systems Design and Implementation, pages 87–102. USENIX Association, 2000.

[12] T. M. Madhyastha and D. A. Reed. Input/output access pattern classification using hidden Markov models. Workshop on Input/Output in Parallel and Distributed Systems, pages 57–67. ACM Press, December 1997.

[13] S. McFarling. *Combining branch predictors*. Technical Report TN-36. Digital Western Research Lab., 1993.

[14] T. M. Mitchell. *Machine learning*. McGraw-Hill, 1997.

[15] S. W. Ng. Improving disk performance via latency reduction. *IEEE Transactions on Computers*, **40**(1):22–30, January 1991.

[16] J. Oly and D. A. Reed. Markov model prediction of I/O requests for scientific applications. Proceedings of the 16th international conference on Supercomputing. ACM Press, 2002.

[17] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, **10**(1):26–52. ACM Press, February 1992.

[18] C. Ruemmler and J. Wilkes. UNIX disk access patterns. Winter USENIX Technical Conference, pages 405–420, 1993.

[19] C. Ruemmler and J. Wilkes. *Disk Shuffling*. Technical report HPL-91-156. Hewlett-Packard Company, Palo Alto, CA, October 1991.

[20] S. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Prentice Hall, December 2002.

[21] J. Schindler and G. R. Ganger. *Automated disk drive characterization*. Technical report CMU–CS–99–176. Carnegie-Mellon University, Pittsburgh, PA, December 1999.

[22] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned extents: matching access patterns to disk drive characteristics. Conference on File and Storage Technologies, pages 259–274. USENIX Association, 2002.

[23] E. Shriver, E. Gabber, L. Huang, and C. A. Stein. Storage management for web proxies. USENIX Annual Technical Conference, pages 203–216, 2001.

[24] T-YYeh and Y. N. Patt. Two-level adaptive branch prediction. 24th ACM/IEEE International Symposium and Workshop on Microarchitecture, pages 51–61, 1991.

[25] J. Wang and Y. Hu. PROFS – Performance-Oriented Data Reorganization for Log-structured File System on Multi-Zone Disks. Ninth International Symposium on Modeling, Analysis and Simulation on Computer and Telecommunication Systems, pages 285–293, 2001.

[26] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson. Trading capacity for performance in a disk array. Symposium on Operating Systems Design and Implementation, pages 243–258. USENIX Association, 2000.